MAKING CRYPTO LIBRARIES ROBUST AGAINST PHYSICAL SIDE-CHANNEL ATTACKS

A Dissertation
Presented to
The Academic Faculty

By

Md Monjur Alam

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2019

MAKING CRYPTO LIBRARIES ROBUST AGAINST PHYSICAL SIDE-CHANNEL ATTACKS

Approved by:

Dr. Milos Prvulovic, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Alenka Zajic
School of Computer Science
Georgia Institute of Technology

Dr. Alexandra Boldyreva
School of Computer Science
Georgia Institute of Technology

Dr. Raheem Beyah
School of Computer Science
Georgia Institute of Technology

Dr. Angelos Keromytis
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: September 6, 2019

I dedicate this thesis to my father.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

The connection between theoretical and applied cryptography is often not well established due to difficulties in translating the theoretical security proofs to real world software and hardware implementations. Cryptanalysis is an important branch in cryptology and focuses on breaking cryptographic primitives and protocols. Side-channel analysis is a cryptanalytic technique born from practice. Physical side-channel cryptanalysis is a very effective approach to break a secure cryptographic system. These attacks are based on signals generated from a processor while it carries out computations. These signals include electromagnetic (EM) emanations created by current flow within the device's computational and power-delivery circuitry [1, 2, 3, 4, 5, 6], variation in power consumption [7, 8, 9, 10, 11, 12, 13, 14, 15, 16], sound [17, 18, 19, 20], and also temperature [21, 22].

Most side-channel attacks on cryptographic primitives and implementations rely on different control flow or memory access patterns. As a countermeasure, the cryptographic community has established the notion of *constant time program code*, which is designed to offer protection from microarchitectural attacks [23, 24], and also from simple single-trace power analysis attacks [25, 26, 27]. At a high level, *constant time program code* is implemented such that the execution time, control-flow, and memory access pattern, are all independent of the value of the program's secrets from attacker-observable leakage sources. This notion of security is standardized [28] and enforced by the cryptographic implementations, i.e., `OpenSSL`, `Libgcrypt`, etc.

Overall, existing implementations of cryptographic primitives are designed to mitigate previously known physical and cache-based side-channel attacks by removing key-dependent timing variation and large operand-dependent changes in overall activity during big-number operations, e.g. use of low Hamming-weight operands in large-number multi-

plication operations during RSA modular exponentiation. However, existing cryptographic implementations do not consider operand-dependent activity during brief operations, such as reading a bit of the secret key, possibly because these operations were considered difficult to precisely identify and/or exploit in attacks.

While there are various optimization considerations and counter-measures enforced for the public key cryptographic primitives and implementations, there exist some signal differences created by systematic differences in operand values during secret value computation. Capturing and identifying emanated signals while processing this sensitive information, although very minute, make it possible to break the secret key.

This information leakage is hard to detect by software developers as it requires high level of expertise on the hardware design of computer systems and compiler optimization. Additionally, since discovering or detecting side-channel information leakage is not part of the standard developer practices, it is hard to detect vulnerabilities due to side/covert channels. Nevertheless, this type of attack represents a serious and immediate threat to the security of Internet societies who use these primitives.

The research presented here is motivated by the ongoing and young field of research trying to exploit leakage of side-channel information to mount key recovery attacks against software implementations of cryptographic primitives that are believed to be secure. This thesis focuses on detailing a set of new techniques to exploit widely used open sources for software implementations of cryptographic primitives. The work embraces the importance of re-thinking before designing and implementing public key cryptography (PKC), in general. It was widely believed that constant time program code does not produce easily-observable operand-dependent physical leakage and is protected from the analysis based on power and EM emanations. This thesis primarily addresses the following questions: (1) Is constant time program code resilient to power and EM analyses? (2) How can an attacker extract a secret key from such an implementation?

We observe that, while a constant-time implementation ensures that the control-flow

and cache-block access sequence is the same regardless of the secret key, that results in very stable timing that can be used to very precisely locate (in the overall timeline of the signal) the signal snippet that corresponds to a specific brief part of the computation, e.g. the signal that corresponds to operations on individual bits of the secret key t. If, as we have found in several cryptographic implementations, that brief part of the computation considers only one bit of the key (at a time), and if the two possible values of the bit result in even a very small (in magnitude) and very brief (in time) difference in side-channel signals, the bit is at risk of being recovered from the signal using a binary (only two possible decisions) classifier.

This thesis exhibits that constant-time implementations of cryptographic primitives can be vulnerable against physical side-channel EM attacks.

1.1 Research Contributions

The main overall contributions of this thesis are as follows:

1. We demonstrate that snippets of the signal that are of interest for cryptanalysis can be identified very precisely given the overall signal that corresponds to the entire cryptographic operation, e.g. a decryption or a signing operation.

2. We demonstrate that, given the signal snippets in which individual bits of the key are used as operands, the small-magnitude and short-duration key-bit-dependent changes in the signal are sufficient for a binary (only two possible decisions) classifier to extract the value of that bit.

3. We propose mitigations that are intended to force the classifier to make decisions among many-possibilities (rather than binary decisions). We achieve this by refactoring the code, when possible, to operate on groups of secret bits rather than individual bits, and by introducing randomization of secret-dependent operands.

1.2 One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA

Prior physical side-channel attacks on PKC implementations rely on classifying signals corresponding to a large-integer square and multiply operations [29, 30, 31]. The focus on identifying this long-lasting subsequence was to identify the overall sequence of samples corresponding to the entire exponentiation. The classification of these long-lasting square-multiplications would be difficult when the sequence of square-multiplications are not key dependent and when the attacker cannot control the input message that would be exponentiated.

Keeping this in mind, `OpenSSL` implements constant-time Montgomery ladder to counter the square-multiply sequence identification. To counter cache timing attacks, OpenSSL enforces scatter-gather technique where a random cache space is used to fetch pre-computed values. These techniques prevent the key exploitation presented in [29, 30, 31] when OpenSSL computes constant-time window exponentiation.

`One&Done` presents a side-channel attack that is based on the analysis of signals that correspond to the brief computation activity that computes the value of each window during exponentiation, i.e. activity *between* large-integer multiplications. A uniform pattern of multiplication computation helps us locate the exact position of each edges of that multiplication and reading a single bit from the key helps us classify the bit computation of zero and one. Most prior works focused on the large-integer multiplications themselves and/or the table lookups that obtain the multiplicand for the computed window value. The values these computations operate on are related to the individual bits of the secret exponent and not the message (cipher text). This absence of message-induced variation allows the small variation caused by different values of an individual exponent bit to "stand out" in the signal and be accurately matched to signals from training. More importantly, this message-independence makes the new attack completely immune to existing countermeasures that

focus on thwarting chosen-ciphertext attacks and/or square/multiply sequence analysis.

To mitigate the side-channel vulnerability exposed by our attack approach, instead of reading a single secret bit, we read a branch of bits which makes 32 choices for 5-bit window length rather than the 2 choices before the mitigation. We change the window value computation to obtain a full integer's worth of bits from the exponent, then mask that value to obtain the window value, rather than constructing the window value one bit at a time with large-number Montgomery multiplication between these one-bit window-value updates. This mitigation causes the signal variation during the brief window computation to depend on tens of bits of the exponent as a group, i.e. the signal variation introduced by one bit in the exponent during the window computation is now superimposed to the variation introduced by the other bits in the group, instead of having each bit's variation alone in its own signal snippet. Our experiments show that this mitigation actually improves exponentiation performance slightly and, more importantly, with this mitigation the recovery rate for the exponents bits becomes equivalent to random guessing.

1.3 Nonce@Once: A Single-Trace EM Side Channel Attack on ECDSA in GnuPG/-libgcrypt and OpenSSL

To mitigate side-channel attacks, in recent versions of cryptographic packages, such as `GnuPG/libgcrypt` and `OpenSSL`, point and message blinding is applied prior to performing point-scalar multiplication, to prevent chosen-message and chosen-base-point attacks. The point-scalar multiplication itself is implemented such that the execution time, control-flow, and memory access pattern, are all independent of the value of the (secret) scalar.

`Nonce@Once` presents a side-channel attack that recovers the secret ECDSA key by analyzing the signal that corresponds to a single ECDSA signing operation in the current (as of this writing) versions of `libgcrypt` and `OpenSSL`. In this implementation, the end of an add-double operation is clearly and precisely identifiable in the signal, and the constant-

time nature of the implementation implies that the program activity that follows, which contains activity that corresponds to reading the next bit of the key from an array, can also be precisely located in the timeline of the signal. Furthermore, since the implementation reads one bit of the key at a time, a binary classifier can be used to determine, for each such snippet of the signal, the value of the secret bit that was used. Specifically, our attack 1) identifies the signal snippet that corresponds to each instance of the conditional swap operation, 2) determines which of the two possible values of the swap condition each signal snippet corresponds to, 3) uses the values of the swap conditions, which directly correspond to individual bits of the secret nonce in an ECDSA signing operation, to construct the set of possible nonce values and 4) reconstructs the full private/public ECDSA key pair.

Finally, we propose a mitigation based on randomizing the exclusive-or mask in the conditional swap operation, which avoids creating a systematic condition-dependent difference in operand values for exclusive-or operations during the conditional swap. We have confirmed that this mitigation is effective in preventing this and similar attacks, and are currently working on submitting this mitigation to both `GnuPG` and `OpenSSL`, and expect the mitigation to be merged into both source code repositories prior to publication of this work.

1.4 Rethinking DSA Implementations: A New Attack Method With A Few EM Traces

While various optimization and counter-measures are enforced for public key cryptography (PKC) implementations, there exist some so called ”glue-codes” which exhibit key dependency. Capturing emanated signals while executing these ”glue-codes” can lead to break PKC implementations.

In that respect, we present a new physical side-channel attack on PKC implementations of OpenSSL. In particular, we consider DSA implementation as a use case, which utilizes constant-time fixed-window (m-ary) modular exponentiation. A uniform pattern of

constant-time multiplication helps us locate the exact position of each edge of that multiplication (phase I of the attack) and reading a single bit from the key helps us to predict whether that bit is zero or one (phase II of the attack). Our attack is based on the electromagnetic (EM) emissions generated while the device performs cryptographic operations, such as digital signature and verification. We have observed that there exist meaningful EM emissions during "Window" computation for DSA sliding and fixed window implementations. We can split the proposed attack into two phases as training and detection. In the first phase, we train our system using different per-message keys, which are captured EM images of window computations for different keys followed by processing the EM signals in the time-domain. In the detection phase, with a single trace, we compare the images of window computations of test signals with trained signals. We found that our approach can detect 99% exponent bits for constant time DSA implementations. We also propose some counter-measures to hinder this vulnerability. With the proposed counter-measures implemented on the current OpenSSL, we could not detect more than 50% bits for fixed-window. We demonstrated different implementation aspects and their effects as countermeasures which embrace the importance of re-thinking before designing and implementing PKC, in general. To evaluate the robustness and effectiveness of our attacks and corresponding counter-measures, we ported the latest version(at the time of our experiment) of OpenSSL (1.1.0g) to two cell phones and one embedded device, and executed applications that use the OpenSSL sign/verify libraries. We demonstrate that our approach works across different devices.

1.5 NAF Based OpenSSL and its Vulnerability

Before switching to using constant-time implementations of point multiplication during ECDSA signing, OpenSSL used to use NAF based point multiplication. This work presents a side-channel attack that recovers the secret ephemeral secret scalar (nonce) used in the elliptic curve point multiplication by a scalar in ECDSA, using the signal that corresponds

to a single signing operation of `OpenSSL`. In this work we have shown that NAF based implementation is completely exploitable, where 100% of secret nonce can be retrieved. We further have shown that wNAF (where w = 3) based implementation is partially exploitable, where 70% of secret nonce can be retrieved.

1.6 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 describes One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA. In Chapter 3 we present Nonce@Once: A Single-Trace EM Side Channel Attack on Several Constant-Time Elliptic Curve Implementations. We demonstrate how constant time ECC program code resilient to simple EM analysis. Chapter 4 describes attack on DSA and different implementation aspects and their effects as countermeasures. Chapter 5 describes the vulnerabilities on NAF based OpenSSL implementations.

CHAPTER 2

ONE&DONE: A SINGLE-DECRYPTION EM-BASED ATTACK ON OPENSSL'S CONSTANT-TIME BLINDED RSA

2.1 Abstract

This chapter presents the first side channel attack approach that, without relying on the cache organization and/or timing, retrieves the secret exponent from a single decryption on arbitrary ciphertext in a modern (current version of OpenSSL) fixed-window constant-time implementation of RSA. Specifically, the attack recovers the exponent's bits during modular exponentiation from analog signals that are unintentionally produced by the processor as it executes the constant-time code that constructs the value of each "window" in the exponent, rather than the signals that correspond to squaring/multiplication operations and/or cache behavior during multiplicand table lookup operations. The approach is demonstrated using electromagnetic (EM) emanations on two mobile phones and an embedded system, and after only one decryption in a fixed-window RSA implementation it recovers enough bits of the secret exponents to enable very efficient (within seconds) reconstruction of the full private RSA key.

Since the value of the ciphertext is irrelevant to our attack, the attack succeeds even when the ciphertext is unknown and/or when message randomization (blinding) is used. Our evaluation uses signals obtained by demodulating the signal from a relatively narrow band (40 MHz) around the processor's clock frequency (around 1GHz), which is within the capabilities of compact sub-$1,000 software-defined radio (SDR) receivers.

Finally, we propose a mitigation where the bits of the exponent are only obtained from an exponent in integer-sized groups (tens of bits) rather than obtaining them one bit at a time. This mitigation is effective because it forces the attacker to attempt recovery of tens

of bits from a single brief snippet of signal, rather than having a separate signal snippet for each individual bit. This mitigation has been submitted to OpenSSL and was merged into its master source code branch prior to the publication of this chapter.

2.2 Motivation

Side channel attacks extract sensitive information, such as cryptographic keys, from signals created by electronic activity within computing devices as they carry out computation. These signals include electromagnetic emanations created by current flows within the device's computational and power-delivery circuitry [1, 2, 3, 4, 5, 6], variation in power consumption [7, 8, 9, 10, 11, 12, 13, 14, 15, 16], and also sound [17, 18, 19, 20], temperature [21, 22], and chasis potential variation [11] that can mostly be attributed to variation in power consumption and its interaction with the system's power delivery circuitry. Finally, not all side channel attacks use analog signals: some use faults [32, 33], caches [34, 35, 36], branch predictors [37], etc.

Most of the research on physical side-channel attacks has focused on relatively simple devices, such as smartcards and simple embedded systems, where the side-channel signals can be acquired with bandwidth much higher than the clock rates of the target processor and other relevant circuitry (e.g. hardware accelerators for encryption/decryption), and usually with highly intrusive access to the device, e.g. with small probes placed directly onto the chip's package [38, 14]. Recently, attacks on higher-clock-rate devices, such as smartphones and PCs , have been demonstrated [39, 40, 4, 41]. They have shown that physical side channel attacks are possible even when signals are acquired with bandwidth that is much lower than the (gigahertz-range) clock rates of the processor, with less-intrusive access to the device, and even though advanced performance-oriented features, such as super-scalar (multiple instructions per cycle) execution and instruction scheduling, and system software activity, such as interrupts and multiprocessing, cause significant variation in both shape and timing of the signal produced during cryptographic activity.

To overcome the problem of low bandwidth and variation, successful attacks on high-clock-rate systems tend to focus on parts of the signal that correspond to activity that takes many processor cycles. A representative example of this is decryption in RSA, which consists of modular exponentiation of the ciphertext with an exponent that is derived from the private key. The attacker's goal is to recover enough bits of that secret exponent through side-channel analysis, and use that information to compute the remaining parts of the secret key. Most of the computational activity in large-integer modular exponentiation is devoted to multiplication and squaring operations, where each squaring (or multiplication) operation operates on large integers and thus takes many processor cycles.

Prior physical side-channel attacks on RSA rely on classifying the signals that correspond to large-integer square and multiply operations that together represent the vast majority of the computational work when performing large-integer exponentiation [42, 40, 11, 19]. Between these long-lasting square and multiply operations are the few processor instructions that are needed to obtain the next bit (or group of bits) of the secret exponent and use that to select whether the next large-integer operation will be squaring or multiplication, and/or which operands to supply to that operation. The focus on long-lasting operations is understandable, given that side channel attacks ultimately recover information by identifying the relevant sub-sequences of signal samples and assessing which of the possible categories is the best match for each sub-sequence. The sub-sequences that correspond to large-integer operations produce long sub-sequences of samples, so they 1) are easier to identify in the overall sequence of samples that corresponds to the entire exponentiation, and 2) provide enough signal samples for successful classification even when using relatively low sampling rates.

However, the operands in these large-integer operations are each very regular in terms of the sequence of instructions they perform, and the operands used in those instructions are ciphertext-dependent, so classification of signals according to exponent-related properties is difficult unless 1) the sequence of square and multiply operations is key-dependent or 2)

the attacker can control the ciphertext that will be exponentiated, and chooses the ciphertext in a way that produces systematically different side channel signals for each of the possible exponent-dependent choices of operands.

2.2.1 Our Contributions

In this chapter we present a side-channel attack that is based on analysis of signals that correspond to the brief computation activity that computes the value of each window during exponentiation, i.e. activity *between* large-integer multiplications, in contrast to most prior work that focuses on the large-integer multiplications themselves and/or the table lookups that obtain the multiplicand for the computed window value. The short duration of these window value computations may hinder signal-based classification to some extent. However, the values these computations operate on are related to the individual bits of the secret exponent and not the message (ciphertext). This absence of message-induced variation allows the small variation caused by different values of an individual exponent bit to "stand out" in the signal and be accurately matched to signals from training. More importantly, this message-independence makes the new attack completely immune to existing countermeasures that focus on thwarting chosen-ciphertext attacks and/or square/multiply sequence analysis.

The experimental evaluation of our attack approach was performed on two Android-based mobile phones and an embedded system board, all with ARM processors operating at high (800 MHz to 1.1 GHz) frequencies, and the signal is acquired in the 40 MHz band around the clock frequency, resulting in a sample rate that is $<5\%$ of the processor's clock frequency, and well within the signal capture capabilities of compact commercially available sub-$1,000 software-defined radio (SDR) receivers such as the Ettus B200-mini. The RSA implementation we target is the constant-time fixed-window implementation used in OpenSSL [43] version 1.1.0g, the latest version of OpenSSL at the time this chapter was written. Our results show that our attack approach correctly recovers between 95.7% and

99.6% (depending on the target system) of the secret exponents' bits from the signal that corresponds to *a single instance of RSA decryption*, and we further verify that the information from each instance of RSA encryption/signing in our experiments was sufficient to quickly (on average $<$1 second of execution time) fully reconstruct the private RSA key that was used.

To further evaluate our attack approach, we apply it to a sliding-window implementation of modular exponentiation in OpenSSL – this was the default implementation in OpenSSL until Percival et al. [44] demonstrated that its key-dependent square/multiply sequence makes it vulnerable to side channel attacks. We show that in this implementation our approach also recovers nearly all of the secret-exponent bits from a single use (exponentiation) of that secret exponent.

To mitigate the side-channel vulnerability exposed by our attack approach, we change the window value computation to obtain a full integer's worth of bits from the exponent, then mask that value to obtain the window value, rather than constructing the window value one bit at a time with large-number Montgomery multiplication between these one-bit window-value updates. This mitigation causes the signal variation during the brief window computation to depend on tens of bits of the exponent as a group, i.e. the signal variation introduced by one bit in the exponent during the window computation is now superimposed to the variation introduced by the other bits in the group, instead of having each bit's variation alone in its own signal snippet. Our experiments show that this mitigation actually improves exponentiation performance slightly and, more importantly, that with this mitigation the recovery rate for the exponents bits becomes equivalent to random guessing. This mitigation has been submitted to OpenSSL and was merged into its master source code branch on May 30th, 2018, prior to the publication of this chapter.

2.2.2 Threat Model

Assumptions

Our attack model assumes that there is an adversary who wishes to obtain the secret key used for RSA-based public-key encryption or authentication. We further assume that the adversary can bring a relatively compact receiver into close proximity of the system performing these RSA secret-key operation, for example a smart-infrastructure or smart-city device which uses public key infrastructure (PKI) to authenticate itself and secure its communication over the Internet, and which is located in a public location, or that the adversary can hide a relatively compact receiver in a location where systems can be placed in close proximity to it, e.g. under a cellphone charging station at a public location, under the tabletop surface in a coffee shop, etc.).

We assume that the adversary can access another device of the same type as the one being attacked, which is a highly realistic assumption in most attack scenarios described above, and perform RSA decryption/authentication with known keys in preparation for the attack. Unlike many prior attacks on RSA, we do **not** assume that the adversary can choose (or even know) the message (ciphertext for RSA decryption) to which the private key will be applied, and we further assume that the RSA implementation under attack **does utilize blinding** to prevent such chosen-ciphertext attacks. Finally, we assume that it is highly desirable for the attacker to recover the secret key after only very few uses (ideally only one use) of that key on the target device. This is a very realistic assumption because PKI is typically used only to set up a secure connection, typically to establish the authenticity of the communication parties and establish a symmetric-encryption session key, so in scenarios where the attacker's receiver can only be in close proximity to the target device for a limited time, very few uses of the private RSA key may be observed.

Targeted Software

The software we target is OpenSSL version 1.1.0g [43], the latest version of OpenSSL at the time this chapter was written. Its RSA decryption uses constant-time fixed-window large-number modular exponentiation to mitigate both timing-based attacks and attacks that exploit the exponent-dependent variation in the square-multiply sequence. The lookup tables used to update the result at the end of each window are stored in scattered form to mitigate attacks that examine the cache and memory behavior when reading these tables, and the RSA implementation supports blinding (which we turn on in our experiments) to mitigate chosen-ciphertext attacks.

Targeted Hardware

The hardware we target are two modern Android-based smartphones and a Linux-based embedded system board, all with ARM processor clocked at frequencies around 1GHz. In our experiments we place probes very close, but without physical contact with the (un-opened) case of the phone, while for the embedded system board we position the probes 20 cm away from the board, so we consider the demonstrated attacks close-proximity but non-intrusive.

Current Status of Mitigation

The mitigation described in this chapter has been submitted as a patch for integration into the main branch of OpenSSL. This patch was merged into the "master" branch of OpenSSL's source code on May 20th, 2018, before this chapter was published.

2.3 Background

Long-lasting operations (such as large-integer square and multiply operations) facilitate matching by producing numerous signals samples even when the signal is collected at a

```
1  // result r starts out as 1
2  BN_one(r);
3  // For each bit of exponent d
4  for (b=bits-1;b>=0;b--){
5      // r = r*r mod m
6    BN_mod_mul(r,r,r,m);
7    if(BN_is_bit_set(d,b))
8        // r = r*c mod m
9      BN_mod_mul(r,r,c,m);
10 }
```

Figure 2.1: A simple implementation of large-number modular exponentiation

limited sample rate.

A representative example is RSA's decryption, which at its core performs modular exponentiation of the ciphertext c with a secret exponent (d) modulo m or, in more a efficient implementation that rely on the Chinese Reminder Theorem (CRT), two such exponentiations, with secret exponents d_p and d_q with modulo p and q, respectively. The side-channel analysis thus seeks to recover either d or, in CRT-based implementations, d_p and d_q, using side-channel measurements obtained while exponentiation is performed.

The exponentiation is implemented as either left-to-right (starting with the most significant bits) or right-to-left (starting with the least significant bits) traversal of the bits of the exponents, using large-integer modular multiplication to update the result until the full exponentiation is complete. Left-to-right implementations are more common, and without loss of generality we use c to denote the ciphertext, d for the secret exponent, and m for the modulus. A simple implementation of exponentiation considers one exponent bit at a time, as shown in Figure 2.1, which is adapted from OpenSSL's source code.

The `BN` prefix in Figure 2.1 stands for "Big Number" (i.e. large integer). Each large integer is represented by a vector of *limbs*, where a limb is an ordinary (machine-word-sized) integers. The `BN_is_bit_set(d,b)` function returns the value (0 or 1) of the `b`-th bit of large-integer exponent `d`, which only requires a few processor instructions: compute the index of the array element that contains the requested bit, load that element, then shift and bit-mask to keep only the requested bit. The instructions that implement the loop, the

if statement, and function call/return are also relatively few in number.

However, the `BN_mod_mul` operation is much more time-consuming: it requires numerous multiplication instructions that operate on the limbs of the large-integer multiplicands. Large integers c, d, and m (or, in CRT-based implementations the d_q, d_p and the corresponding moduli), all have $O(n)$ bits and thus $O(n)$ limbs, where n is the size of the RSA cryptographic key. A grade-school implementation of `BN_mod_mul` thus requires $O(n^2)$ limb multiplications, but the Karatsuba multiplication algorithm [45] is typically used to reduces this to $O(n^{log_2 3}) \approx O(n^{1.585})$, In most modern implementations a significant further performance improvement is achieved by converting the ciphertext to a Montgomery representation, using Montgomery multiplication for `BN_mod_mul` during exponentiation, and at the end converting the result r back to the standard representation.

Even with Montgomery multiplication, however, the vast majority of execution time for large-number exponentiation is spent on large-number multiplications, so performance optimizations focus on reducing the number of these multiplications. Likewise, most of the side-channel measurements (e.g. signal samples) collected during large-number exponentiation correspond to large-number multiplication activity, so existing side channel cryptanalysis approaches tend to target multiplication activity.

One class of attacks focuses on distinguishing between squaring ($r * r$) and multiplication ($r * c$) operations, and recovering information about the secret exponent from the sequence in which they occur. Examples of such attacks include FLUSH+RELOAD [46] (which uses instruction cache behavior) and Percival's attack [44], which uses data cache behavior. In the naive implementation above, an occurrence of squaring tells the attacker that the next bit of the exponent is being used, and an occurrence of multiplication indicates that the value of that bit is 1, so an attack that correctly recovers the square-multiply sequence can trivially obtain all bits of the secret exponent.

To improve performance, most modern implementations use window-based exponentiation, where squaring is needed for each bit of the exponent, but a multiplication is needed

only once per a multi-bit group (called a *window*) of exponent bits. A left-to-right (starting at the most significant bit) *sliding-window* implementation scans the exponent bits and forms windows of varying length. Since a window that contains only zero bits requires no multiplication (and thus cannot benefit from forming multi-bit windows), only windows that begin and end with 1-valued bits are allowed to form multi-bit windows, whereas zero bits in-between these windows are each treated as their own single-bit windows that can omit multiplication. A sliding-window implementation is shown in Figure 2.2, using code adapted from OpenSSL's source code for sliding-window modular exponentiation. The sliding-window approach chooses a maximum size $wmax$ for the windows it will use, precomputes a table ct that contains the large-integer value $c^{wval} mod m$ for each possible value $wval$ up to $wmax$ length, and then scans the exponent, forming windows and updating the result for each window.

In this algorithm, a squaring (lines 7 and 26 in Figure 2.2) is performed for each bit while the multiplication operation (line 29) is performed only at the (1-valued) LSB of a non-zero window. Thus the square-multiply sequence reveals where some of the 1-valued bits in the exponent are, and additional bits of the exponent have been shown to be recoverable [42] by analyzing the number of squaring between each pair of multiplications. The fraction of bits that can be recovered from the square-multiply sequence depends on the maximum window size $wmax$, but commonly used values of $wmax$ are relatively small and prior work [42] has experimentally demonstrated recovery of 49% of the exponent's bits on average when $wmax = 4$ based on the square-multiply sequence. Additional techniques [42, 47] have been shown to recover the full RSA private key once enough of the exponent bits are known, and for $wmax = 4$ this has allowed full key recovery for 28% of the keys [42]. Finally, recent work has shown that fine-grained control flow tracking through analog side channels can be very accurate [48]. Because this sliding-window implementation uses each bit of the exponent to make at least one control flow decision, highly accurate control flow reconstruction amounts to discovering the exponent's bits with some

```
1  BN_one(r);
2  wstart=bits-1;
3  while(wstart>=0){
4    if(!BN_is_bit_set(d,wstart)){
5      // Window is 0, square and
6      // begin a new window
7      BN_mod_mul(r,r,r,m);
8      wstart--;
9      continue;
10   }
11   wval=1;
12   w=1;
13   // Scan up to max window length
14   for(i=1;i<wmax;i++){
15     // Don't go below exponent's LSB
16     if(wstart-i<0)
17       break;
18     // If 1 extend window to it
19     if(BN_is_bit_set(d,wstart-i)){
20       wval=(wval<<(i-w+1))+1;
21       w=i;
22     }
23   }
24   // Square result w times
25   for(i=0;i<w;i++)
26     BN_mod_mul(r,r,r,m);
27   // Multiply window's result
28   // into overall result
29   BN_mod_mul(r,r,ct[wval>>1],m);
30   // Begin a new window
31   wstart-=w;
32 }
```

Figure 2.2: Sliding-window implementation of large-number modular exponentiation

probability of error.

Concerns about the exponent-dependent square-multiply sequences have led to adoption of *fixed window* exponentiation in OpenSSL, which combines the performance advantages of window-based implementation with an exponent-independent square-multiply sequence. This implementation is represented in Figure 2.3, again adapted from OpenSSL's source code.

All windows now have the same number of bits `w`, with exactly one multiplication performed for each window – in fact, all of the control flow is now exactly the same regardless

```
1  b=bits-1;
2  while(b>=0){
3        wval=0;
4        // Scan the window,
5        // squaring the result as we go
6        for (i=0;i<w;i++) {
7              BN_mod_mul(r,r,r,m);
8              wval<<=1;
9              wval+=BN_is_bit_set(d,b);
10             b--;
11       }
12       // Multiply window's result
13       // into the overall result
14       BN_mod_mul(r,r,ct[wval],m);
15 }
```

Figure 2.3: Fixed-window implementation of large-number modular exponentiation

of the exponent. Note that the window value (which consists of the bits from the secret exponent) directly determines which elements of `ct` are accessed. These elements are each a large integers, each of which is typically stored as an array or ordinary integers (e.g. OpenSSL's "Big Number" BN structure). Since each such array is much larger than a cache block, different large integers occupy distinct cache blocks, and thus the address the cache set that is accessed when reading the elements of the `ct` array reveals key material. Percival's attack [44], for example, can note the sequence in which the cache sets are accessed by the victim during fixed-window exponentiation, which reveals which window values were used and in what sequence, which in turns yields the bits of the secret exponent. To mitigate such attacks, the implementation in OpenSSL has been changed to store ct such that each of the cache blocks it contains parts from a number of ct elements, and therefore the sequence of memory blocks that are accessed in each `ct[wval]` lookup leak none or very few bits of that lookup's $wval$.

Another broad class of side channel attacks relies on choosing the ciphertext such that the side-channel behavior of the modular multiplication reveals which of the possible multiplicands is being used. For example, Genkin et al. [11, 19] construct a ciphertext that produces many zero limbs in any value produced by multiplication with the ciphertext,

but when squaring such a many-zero-limbed value the result has fewer zero limbs, resulting in an easily-distinguishable side channel signals whenever a squaring operation (`BN_mod_mul(r,r,r,m)` in our examples) immediately follows a 1-valued window (i.e. when r is equal to $r_{prev} * c \ mod \ m$). This approach has been extended [4] to construct a (chosen) ciphertext that reveals when a particular window value is used in multiplication in a windowed implementation, allowing full recovery of the exponent by collecting signals that correspond to 2^w chosen ciphertexts (one for each window value). However, chosen-ciphertext attacks can be prevented in the current implementation of OpenSSL by enabling *blinding*, which combines the ciphertext with an encrypted (using the public key) random "ciphertext", performs secret-exponent modular exponentiation on this *blinded* version of the ciphertext, and then "unblinding" the decrypted result.

Overall, because large-integer multiplication is where large-integer exponentiation spends most of its time, most of the side-channel measurements (e.g. signal samples for physical side channels) also correspond to this multiplication activity and thus both attacks and mitigation tend to focus on that part of the signal, leaving the (comparably brief) parts of the signal in-between the multiplications largely unexploited by attacks but also unprotected by countermeasures. The next section describes our new attack approach that targets the signal that corresponds to computing the value of the window, i.e .the signal *between* the multiplications.

2.4 Proposed Attack Method

In both fixed- and sliding-window implementations, our attack approach focuses on the relatively brief periods of computation that considers each bit of the exponent and forms the window value $wval$. The attack approach has three key components that we will discuss as follows. First, Section 2.4.1 describes how the signal is received and pre-processed. Second, Section 2.4.2 describes how we identify the point in the signal's timeline where each interval of interest begins. Finally, we describe how the bits of the secret exponent are

recovered from these signal snippets for fixed-window (Section 2.4.3) and sliding-window (Section 2.4.4) implementations.

2.4.1 Receiving the Signal

The computation we target is brief and the different values of exponent bits produce relatively small variation in the side-channel signal, so the signals subjected to our analysis need to have sufficient bandwidth and signal-to-noise ratio for our analysis to succeed. To maximize the signal-to-noise ratio while minimizing intrusion, we position EM probes just outside the targeted device's enclosure. We then run RSA decryption in OpenSSL on the target device while recording the signal in a 40 MHz band around the clock frequency. The 40 MHz bandwidth was chosen as a compromise between recovery rate for the bits of the secret exponent and the availability and cost of receivers capable of capturing the desired bandwidth. Specifically, the 40 MHz bandwidth is well within the capabilities of Ettus USRP B200-mini receiver, which is very compact, costs less than $1,000, and can receive up to 56 MHz of bandwidth around a center frequency that can be set between 70 MHz and 6 GHz, and yet the 40 MHz bandwidth is sufficient to recover nearly all bits of the secret exponent from a single instance of exponentiation that uses that exponent.

We then apply AM demodulation to the received signal, and finally upsample it by a factor of 4. The upsampling consists of interpolating through the signal's existing sample points and placing additional points along the interpolated curve. This is needed because our receiver's sampling is not synchronized in any way to the computation of interest, so two signal snippets collected for the same computation may be misaligned by up to half of the sample period. Upsampling allows us to re-align these signals with higher precision, and we found that 4-fold upsampling yields sufficient precision for our purposes.

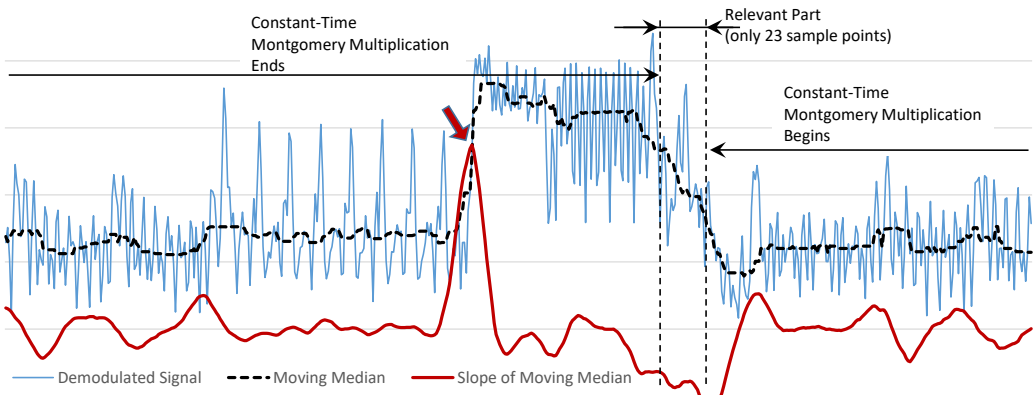

Figure 2.4: Signal that includes the end of one Montgomery multiplication, then the part relevant to our analysis, and then the beginning of another Montgomery multiplication. The horizontal axis is time (from left to right) and the vertical axis is the magnitude of the AM-demodulated signal.

2.4.2 Identifying Relevant Parts of the Signal

Figure 2.4 shows a brief portion of the signal that begins during fixed-window exponentiation in OpenSSL. It includes part of one large-number multiplication (Line 7 in Figure 2.3), which in OpenSSL uses the Montgomery algorithm and a constant-time implementation designed to avoid multiplicand-dependent timing variation that was exploited by prior side-channel attacks. The point in time where Montgomery multiplication returns and the relevant part of the signal begins is indicated by a dashed vertical line in Figure 2.4. In this particular portion of the signal, the execution proceeds to lines 8 and 9 Figure 2.2, where a bit of the exponent is obtained and added to $wval$, then lines 10 and 6, and then 7 where, at the point indicated by the second dashed vertical line, it enters another Montgomery multiplication, whose signal continues well past the right edge of Figure 2.4. As indicated in the figure, the relevant part of the signal is very brief relative to the duration of the Montgomery multiplication.

A naive approach to identifying the relevant snippets in the overall signal would be to obtain reference signal snippets during training and then, during the attack, match against these reference snippets at each position in the signal and use the best-matching parts of the signal. Such signal matching works best when looking for a snippet that has prominent

features, so they are unlikely to be obscured by the noise, and whose prominent features occur in a pattern which is unlikely to exist elsewhere in the signal. Unfortunately, the signal snippets relevant for our analysis have little signal variation (relative to other parts of the signal) and a signal shape (just a few up-and-downs) that many other parts of the signal resemble. In contrast, the signal that corresponds to the Montgomery multiplication has stronger features, and they occur in a very distinct pattern.

Therefore, instead of finding instances of relevant snippets by matching them against their reference signals from training, we use as a reference the signal that corresponds to the most prominent change in the signal during Montgomery multiplication, where the signal abruptly changes from a period with a relatively low signal level to a period with a relatively high signal level. We identify this point in the signal using a very efficient algorithm. We first compute the signal's moving median (thick dashed black curve in Figure 2.4) to improve resilience to noise. We then examine the derivative (slope) of this moving median (thick red curve in Figure 2.4) to identify peaks that significantly exceed its statistically expected variation. In Figure2.4 the thick red arrow indicates such a peak, which corresponds to the most prominent change in the Montgomery multiplication that precedes the relevant part of the signal. Because the implementation of the Montgomery multiplication was designed to have almost no timing variation, the signal snippet we actually need for analysis is at a fixed time offset from the point of this match.

Because this method of identifying the relevant snippets of the signal is based on the signal that corresponds to the Montgomery multiplication that precedes each relevant snippet, the same method can be used for extracting relevant signal snippets for both fixed-window and sliding-window exponentiation – in both cases the relevant snippet is at the (same) fixed offset from the point at which a prominent-enough peak is detected in the derivative of the signal's moving median.

2.4.3 Recovering Exponent Bits in the Fixed-window Implementation

In the fixed-window implementation, large-number multiplication is used for squaring (Line 7 in Figure 2.3) and for updating the result after each window (Line 14). Thus there are four control-flow possibilities for activity between Montgomery multiplications.

The first two control flow possibilities begin when the Montgomery multiplication in line 7 completes. Both control flow possibilities involve updating the window value to include another bit from the exponent (lines 8, 9, and 10), and at line 6 incrementing i and checking it against w, the maximum size of the window. The first control flow possibility is the more common one - the window does not end and the execution proceeds to line 7 when another multiplication at line 7. We label this control flow possibility S-S (from a squaring to a squaring). The second control flow possibility occurs after the last bit of the window is examined and added to $wval$, and in that case the loop at line 6 is exited, the parameters for the result update at line 14 are prepared, and the Montgomery multiplication at line 14 begins. The parameter preparation in our code example would involve computing the address of $ct[wval]$ to create a pointer that would be passed to the Montgomery multiplication as its second multiplicand. In OpenSSL's implementation the ct is kept in a scattered format to minimize leakage of $wval$ through the cache side channel while computing the Montgomery multiplication, so instead the value of $wval$ is used to gather the scattered parts of $ct[wval]$ into a pre-allocated array that is passed to Montgomery multiplication. Since this pre-allocated array is used for all result-update multiplications, memory and cache behavior during the Montgomery multiplication no longer depend on $wval$. This means that in this second control-flow possibility involves significant activity to gather the parts of the multiplicand and place them into the pre-allocated array, and only then the Montgomery multiplication at line 14 begins. We label this control flow possibility S-U (from a squaring to an update).

The last two control flow possibilities occur after the result update in line 14 completes its Montgomery multiplication. The loop condition at line 2 is checked, and then one con-

trol flow possibility (third of the four) is that the entire exponentiation loop exits. We label this control flow possibility U-X (from an update to an exit). The last control-flow possibility, which occurs for all windows except the last one, is that after line 2 we execute line 3, enter the window-scanning loop at line 6, and begin the next large-number Montgomery multiplication at line 7. We label this control flow possibility U-S (from an update to a squaring).

The sequence in which these four control flow possibilities are encountered in each window is always the same: $w - 1$ occurrences of S-S, then one occurrence of S-U, then either U-S or U-X, where U-X is only possible for the last window of the exponent.

The first part of our analysis involves distinguishing among these four control flow possibilities. The reason for doing so is that noise bursts, interrupts, and activity on other cores can temporarily interfere with our signal and prevent detection of Montgomery multiplication. In such cases, sole reliance on the known sequence of control flow possibilities would cause a "slip" between the observed sequence and the expected one, causing us to use incorrect reference signals to recover bits of the exponent and to put the recovered bits at incorrect positions within the recovered exponent.

The classification into the four possibilities is much more reliable than recovery of exponent's bits. Compared to the other three possibilities, S-U spends significantly more time between Montgomery multiplications (because of the multiplicand-gathering activity), so it can be recognized with high accuracy and we use it to confirm that the exponentiation has just completed a window. The U-X possibility is also highly recognizable because, instead of executing Montgomery multiplication after it, it leads to executing code that converts from Montgomery to standard large-number format, and it serves to confirm that the entire exponentiation has ended. The S-S and U-S snippets both involve only a few instructions between Montgomery multiplications so they are harder to tell apart, but our signal matching still has a very high accuracy in distinguishing between them.

After individual snippets are matched to the four possibilities, that matching is used

to find the most likely mapping of the sequence of snippets onto the known valid sequence. For example, if for $w = 5$ we observe S-U, U-S, S-S, S-S, S-S, S-U, all with high-confidence matches, we know that one S-S is missing for that window. We then additionally use timing between these snippets to determine the position of the missing S-S. Even if that determination is erroneous, we will correctly begin the matching for the next window after the S-U, so a missing snippet is unlikely to cause any slips, but even when it does cause a slip, such a slip is very likely to be "contained" within one exponentiation window. Note that a missing S-U or S-S snippet prevents our attack from using its signal matching to recover the value of the corresponding bit. A naive solution would be to assign a random value to that bit (with a 50% error rate among missing bits). However, for full RSA key recovery missing bits (erasures, i.e. the value of the bit is known to be unknown) are much less problematic than errors (the value of the bit is incorrect but not known a priori to be incorrect), we label these missing bits as erasures.

Finally, for S-S and S-U snippets we perform additional analysis to recover the bit of the exponent that snippet corresponds to. Recall that, in both S-S and S-U control flow possibilities, in line 9 a new bit is read from the exponent and is added to $wval$, and that bit is the one we will recover from the snippet. For ease of discussion, we will refer to the value of this bit as $bval$. To recover $bval$, in training we obtain examples of these snippets for each value of $bval$. To suppress the noise in our reference snippets and thus make later matching more accurate, these reference snippets are averages of many "identical" examples from training. Clearly, there should be separate references for $bval = 0$ (where only $bval = 0$ examples are averaged) and for $bval = 1$ (where only $bval = 1$ examples are averaged. However, $bval$ is not the only value that affects the signal in a systematic way – the signal in this part of the computation is also affected by previous value of $wval$, loop counter i, etc. The problem is that these variations occur in the same part of the signal where variations due to $bval$ occur, so averaging of these different variants may result in attenuating the impact of $bval$. We alleviate this problem by forming separate references for different bit-

positions within the window, e.g. for window size $w = 5$ each value of $bval$ would have 4 sets of S-S snippets and one set of S-U snippets, because the first for bits in the window correspond to S-S snippets and the last bit in the window to an S-U snippet. To account for other value-dependent in the signal, in each such set of snippets we cluster similar signals together and use the centroid of each cluster as the reference signal. We use the K-Means clustering algorithm and the distance metric used for clustering is Euclidean distance (sum of squared differences among same-position samples in the two snippets). We found that having at least 6-10 clusters for each set of snippets discussed above improves accuracy significantly. Beyond 6-10 clusters our recovery of secret exponent's bits improves only slightly but requires more training examples to compensate for having fewer examples per cluster (and thus less noise suppression in the cluster's centroid). Thus we use 10 clusters for each window-bit-position for each of the two possible values of $bval$. Overall, the number of S-S reference snippets for $bval$ recovery is $2 * (w - 1) * 10$ – two possible values of $bval$, $w - 1$ bit-positions, 10 reference signals (cluster centroids) for each, while for S-U snippets we only have 20 reference snippets because S-U only happens for the last bit-position in the window. For commonly used window sizes this results in a relatively small overall number of reference snippets, e.g. for $w = 5$ there are only 100 reference snippets. To illustrate the difference in the signals created by the value of the exponent's bit, Figure 2.5 shows two reference S-S snippets (cluster centroids) for each value of the exponent's bit, with the most significant differences between 0-value and 1-value signals indicated by thick arrows.

Recall that, before attempting recovery of an unknown bit of the secret exponent, we have already identified which control-flow possibility (S-S or S-U) the snippet under consideration belongs to, and for S-S which bit-position it belongs to, so there are 20 reference snippets that each snippet-under-consideration is compared to (10 clusters for $bval = 0$ and 10 clusters for $bval = 1$). Thus the final step of our analysis involves finding the closest match (using Euclidean distance as a metric) among these 20 reference snippets and taking

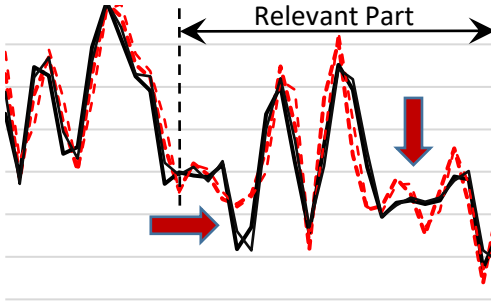

Figure 2.5: Example signal references (cluster centroid) for S-S snippets. Two references are shown for each value of the exponent's bit that corresponds to the snippet.

the $bval$ associated with that reference snippet.

2.4.4 Recovering Exponent Bits in the Sliding-window Implementation

The sliding-window implementation of large-integer exponentiation (Figure 2.2) has three sites where Montgomery multiplication is called: the squaring within a window at line 26, which we label S, the update of the result at line 29, which we label U, and the squaring for a zero-valued window at line 7, which we label Z. The control flow possibilities between these include going from a squaring to another squaring (which we label as S-S). This transition is very brief (it only involves staying in the loop at line 25). The other transitions are S-U, which consumes more time because it performs the $ct[wval]$ computation; U-Z, which involves executing line 31, line 3, line 4 (where a bit of the exponent is examined), and finally entering Montgomery multiplication at line 7; U-S, which involves executing line 31, line 3, line 4, lines 11 and 12, and the entire window-scanning loop at lines 14-23, then line 25 and finally entering Montgomery multiplication at line 26; Z-Z where after line 7 the execution proceeds to line 8, line 9, line 3, line 4, and line 7 again; Z-S where after line 7 the execution proceeds to lines 8, 9, 3, 4, and then to lines 11 and 12, the loop at line 14-23, then line 25 and finally line 26; U-X where after the Montgomery multiplication at line 29 the execution proceeds to line 31 and then exits the loop at line 3; and finally S-X, where after Montgomery multiplication at line 7 the execution proceeds to lines 8 and 9 and then exits the loop at line 3.

Just like in fixed-window implementations, our recovery of the secret exponent begins with determining which snippet belongs to which of these control-flow possibilities. While in Section 2.4.3 this was needed only to correct for missing snippets, in the sliding-window implementation the window size varies depending on which bit-values are encountered in the exponent, so distinguishing among the control-flow possibilities is crucial for correctly assigning recovered bits to bit-positions in the exponent even if no snippets are missing. Furthermore, many of the exponent's bits can be recovered purely based on the sequence of these control-flow possibilities.

Our overall approach for distinguishing among control flow possibilities is similar to that in Section 2.4.3, except that here there are more control-flow possibilities, and the U-S and Z-S coarse-grained possibilities each have multiple control flow possibilities within the snippet: for each bit considered for the window, line 19 determines whether or not to execute lines 20 and 21. However, at the point in the sequence where U-S can occur, the only alternative is U-Z, which is much shorter and thus they are easy to tell apart. Similarly, the only alternative to Z-S is the much shorter Z-Z, so they are also easy to tell apart.

By classifying snippets according to which control-flow possibility they belong (where U-S and U-Z are each treated as one possibility), and by knowing the rules the sequence of these must follow, we can recover from missing snippets and, more importantly, use rules similar to those in [42] to recover many of the bits in the secret exponent. However, in contrast to work in [42] that could only distinguish between a squaring (line 7 or line 26, i.e. S or Z in our sequence notation) and an update (line 29, U in our sequence notation) using memory access patterns within each Montgomery multiplication (which implements both squaring and updates), our method uses the signal snippets between these Montgomery multiplications to recover more detailed information, e.g., for each squaring our recovered sequence indicates whether it is an S or a Z, and this simplifies the rules for recovery of exponent's bits and allows us to extract more of them. Specifically, after a U-S or Z-S, which compute the window value $wval$, the number of bits in the window can be obtained

by counting the S-S occurrences that follow before an S-U is encountered. For example, consider the sequence U-S, S-S, S-S, S-U, U-Z, Z-Z, Z-Z, Z-S. The first U-S indicates that a new window has been identified and a squaring for one of its bits is executed. Then the two occurrences of S-S indicate two additional squaring for this window, and S-U indicates that only these three squaring are executed, so the window has only 3 bits. Because the window begins and ends with 1-valued bits, it is trivial to deduce the values of two of these 3 bits. If we also know that $wmax = 5$, the fact that the window only has 3 bits indicates that the two bits after this window are both 0-valued (because a 1-valued bit would have expanded the window to include it). Then, after S-U, we observe U-Z, which indicates that the bit after the window is 0-valued (which we have already deduced), then two occurrences of Z-Z indicate two more 0-valued bits (one of which we have already deduced), and finally Z-S indicates that a new non-zero window begins, i.e. the next bit is 1. Overall, out of the seven bits examined during this sequence, six were recovered solely based on the sequence. Note that two of the bits (the two zeroes after the window) were redundantly recovered, and this redundancy helps us correct mistakes such as missing snippets or miss-categorized snippets.

In general, this sequence-based analysis recovers all zeroes between windows and two bits from each window. In our experiments, when using $wmax = 5$ this analysis alone on average recovers 68% of the secret exponent's bits, and with using $wmax = 6$, another commonly used value for $wmax$, this analysis alone on average recovers 55% of the exponent's bits. These recovery rates are somewhat higher than what square-update sequences alone enable [42], but recall that in our approach sequence recovery is only the preparation for our analysis of exponent-bit-dependent variation within individual signal snippets.

Since the only bits not already recovered are the "inner" (not the first and not the last) bits of each window, and since U-S and Z-S snippets are the only ones that examine these inner bits, our further analysis only focuses on these. To simplify discussion, we will use U-S to describe our analysis because the analysis for Z-S snippets is virtually identical.

Unlike fixed-window implementations, where the bits of the exponent are individually examined in separate snippets, in sliding-window implementations a single U-S or Z-S snippet contains the activity (line 4) for examining the first bit of the window and the execution of the entire loop (lines 14-23) that constructs the $wval$ by examining the next $wmax - 1$. Since these bits are examined in rapid succession without intervening highly-recognizable Montgomery multiplication activity, it would be difficult to further divide the snippet's signal into pieces that each correspond to consideration of only one bit. Instead, we note that $wmax$ is relatively small (typically 5 or 6), and that there are only 2^{wmax} possibilities for the control flow and most of the operands in the entire window-scanning loop. Therefore, in training we form separate reference snippets for each of these possibilities, and then during the attack we compare the signal snippet under consideration to each of the references, identify the best-matching reference snippet (smallest Euclidean distance), and use the bits that correspond to that reference as the recovered bit values.

2.4.5 Full Recovery of RSA Private Key Using Recovered Exponent Bits

Our RSA key recovery algorithm is a variant of the algorithm described by Henecka et al. [49], which is based on Heninger and Shacham's branch-and-prune algorithm [47]. Like Bernstein et al. [42], we recover from the side channel signal only the bits of the private exponents d_p and d_q, and the recovery of the full private key relies on exploiting the numerical relationships (Equations (1) in Bernstein et al. [42]) between these private exponents (d_p and d_q), the public modulus N and exponent e, and p and q, the private factors of N:

$ed_p = 1 + k_p(p-1) \ mod \ 2^i$

$ed_q = 1 + k_q(q-1) \ mod \ 2^i$

$pq = N \ mod \ 2^i$

where k_p and k_q are positive integers smaller than the public exponent e and satisfy $(k_p - 1)(k_q - 1) \equiv k_p k_q N \ mod \ e$. The public exponent practically never exceeds 32 bits [47] and in most cases $e = 65537$, so a key recovery algorithm needs to try at most e pairs of k_p, k_q.

We could not simply apply Bernstein's algorithm [42] to the exponents recovered by our signal analysis because, like the original branch-and-prune algorithm, such recovery requires certain knowledge of the bit values at some fraction of bit-positions in d_p and d_q, while the remaining bits are unknown but *known to be unknown*, i.e. they are *erasures* rather than errors. Such branch-and-prune search has been shown to be efficient when up to 50% of the bit-positions (chosen uniformly at random) in d_p and d_q are erasures, while its running time grows exponentially when the erasures significantly exceed 50% of the bit positions.

Henecka's algorithm [49] can be applied with the above pruning equations to recover the private key when some of the bits are in error. However, its pruning is based on a key assumption that errors are uniformly distributed, and it does not explicitly consider erasures. Recall, however, that for some of the bit positions our analysis cannot identify the relevant signal snippet for matching against training signals (see Section 2.4.2), which results in an erasure. A naive approach for handling erasures would be to randomly assign a bit value for each erasure (resulting in a 50% error rate among erasures) and then apply Henecka's algorithm. Unfortunately, the erasures during our recovery are a product of disturbances in the signal that are very large in magnitude, and such a disturbance also tends to last long enough to affect multiple bits. With random values assigned to erasures, this produces 50%-error-rate bursts that are highly unlikely to be produced by uniformly distributed errors, causing Henecka's algorithm to either prune the correct partial candidate key or become inefficient (depending on the choice of the ϵ parameter).

Instead, we modify Henecka's algorithm to handle erasures by branching at a bit position when it encounters an erasure, but ignoring that bit position for the purposes of making a pruning decision. We further extend Henecka's algorithm to not do a "hard" pruning of a candidate key when its error count is too high. Instead, we save such a candidate key so that, if no candidate keys remain but the search for the correct private key is not completed, we can "un-prune" the lowest-error-count candidate keys that were previously pruned due

to having too high of an error count. This is similar to adjusting the value of ϵ in Henecka's algorithm and retrying, except that the work of previous tries is not repeated, and this low cost of relaxing the error tolerance allows us to start with a low error tolerance (large ϵ in Henecka et al.) and adjust it gradually until the solution is found.

We further modify Henecka's algorithm to, rather than expand a partial key by multiple bits (parameter t in Henecka et al.) at a time, expand by one bit at a time and, among the newly created partial keys, only further expand the lowest-recent error-count ones until the desired expansion count (t) is reached. In Henecka's algorithm, full expansion by t bits at a time creates 2^t new candidate keys, while our approach discovers the same set of t-times-expanded non-pruned candidates without performing all t expansions on those candidates that encounter too many errors even after fewer than t single-bit expansions. For a constant t, this reduces the number of partial keys that are examined by a constant factor, but when the actual error rate is low this constant factor is close to 2^t.

Overall, our actual implementation of this modified algorithm is very efficient - it considers (expands by one bit) about 300,000 partial keys per second using a single core on recent mobile hardware (4th generation Surface Pro with a Core i7 processor), and for low actual error rates typically finds a solution after only a few thousand partial keys are considered. We evaluate its ability to reconstruct private RSA keys using d_p and d_q bits that contain errors and/or erasures by taking 1,000 RSA keys, introducing random errors, random erasures, and a half-and-half mix of errors and erasures, at different error/erasure rates, and counting how many partial keys had to be considered (expanded by a bit) before the correct private key was reconstructed. The median number of steps for each error/erasure rate is shown in Figure 2.6. We only show results for error/erasure rates up to 10% because those are the most relevant to our actual signal-based recovery of the exponent's bits.

We observe that our implementation of reconstruction quickly becomes inefficient when only errors are present and the error rate approaches 7%, which agrees with the theoretical results of Henecka et al. – since d_p and d_q are used, the m factor in Henecka et al. is 2,

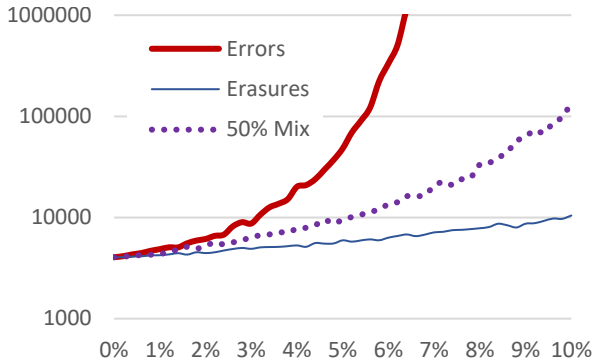

Figure 2.6: Single-bit expansion steps needed to reconstruct the private RSA key (vertical axis, note the logarithmic scale) as a function of the rate at which errors and/or erasures are injected (horizontal axis).

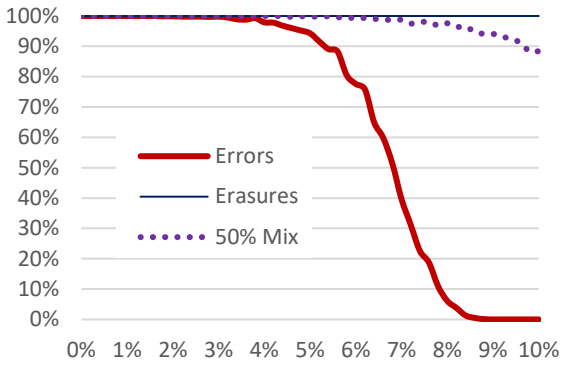

Figure 2.7: Percentage of keys recovered in fewer than 5,000,000 single-bit expansion steps (vertical axis) as a function of the rate at which errors and/or erasures are injected (horizontal axis).

and the upper bound for efficient reconstruction is at 8.4% error rate. In contrast, when only erasures are present, our implementation of reconstruction remains very efficient even as the erasure rate exceeds 10%, which agrees with Bernstein et al.'s finding that reconstruction should be efficient with up to 50% erasure rates. Finally, when equal numbers of errors and erasures are injected, the efficiency for each injection rate is close to (only slightly worse than) the efficiency for error-only injection at half that rate, i.e. with a mix of errors and erasures, the efficiency of reconstruction is largely governed by the errors.

Figure 2.7 shows the percentage of experiments in which the correct RSA key was recovered in fewer than 5,000,000 steps (about 17 seconds on the Surface 4 tablet). When only errors are present, $< 90\%$ of the reconstructions take fewer than 5,000,000 steps

until the error rate exceeds 5.4%, at which point the percentage of under-five-million-steps reconstructions rapidly declines and drops below 10% at the 7.9% error rate. In contrast, all erasure-only reconstructions are under 5,000,000 steps even at the 10% erasure rate. Finally, when erasures and errors are both present in equal measure, the percentage of under-5,000,000-step reconstructions remains above 90% until the injection rate reaches 9.8% (4.9% of the bits are in error and another 4.9% are erased).

2.5 Evaluation

In this section we describe our measurement setup and obtained results for recovering keys from blinded RSA encryption runs on three different devices.

2.5.1 Experimental Setup

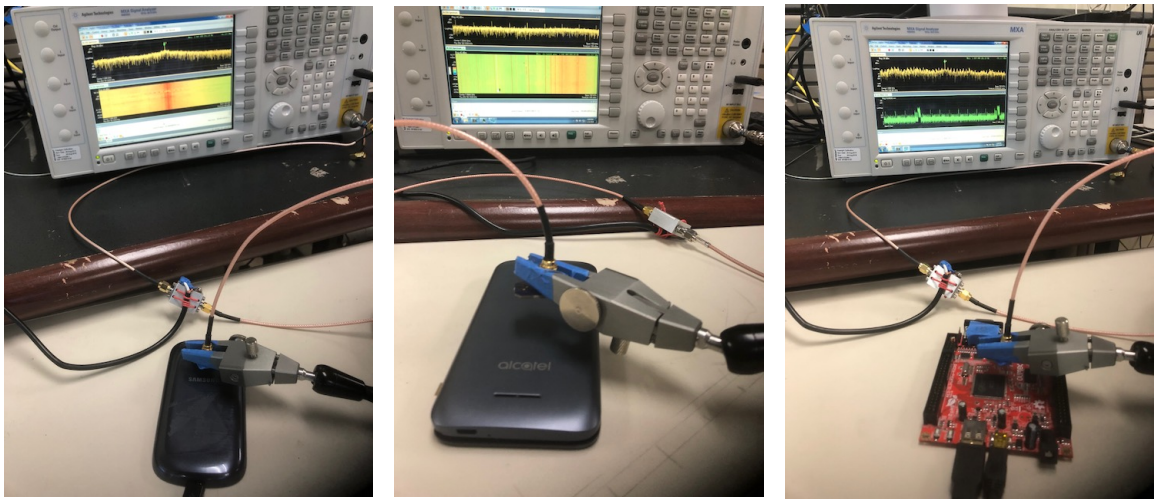

Figure 2.8: The measurement setup for each of the three devices (shown in the right-to-left order): Samsung Galaxy Centura SCH-S738C smart phone, Alcatel Ideal smart phone, and the A13-OLinuXino board.

We run the OpenSSL RSA application on Android smart phones Samsung Galaxy Centura SCH-S738C [50] and Alcatel Ideal [51], and on an embedded device (A13-OLinuXino board [52])) . The Alcatel Ideal cellphone has quad-core 1.1 GHz Qualcomm Snapdragon

processor with Android OS(version 6) and the Samsung phone has a single-core 800 MHz Qualcomm MSM7625A Chipset with Android OS(version 5). The A13- OLinuXino board is a single-board computer that has an in order, 2-issue Cortex A8 ARM processor [53] and runs Debian Linux operating system.

In our experimental setup, we receive signals using small magnetic probe. We place the probe close to the monitored system as shown in Figure 2.8. The signals collected by the probe are recorded with Keysight N9020A MXA spectrum analyzer [54]. Our decision to use spectrum analyzer was mainly driven by its existing features such as built-in support for automating measurements, saving and analyzing measured results, visualizing the signals when debugging code, etc. We have observed very similar signals when using less expensive equipment such as Ettus USRP B200-mini receiver [55]. The analysis was implemented in MATLAB and on a personal computer runs in under one minute per decryption instance (i.e. per recovered 1024-bit exponent).

2.5.2 Experimental Results

2.5.3 Results for OpenSSL's Constant-Time Fixed-Window Implementation

Our first set of experiments evaluates the attack's ability to recover bits of the 1024-bit secret exponent d_p used during RSA-2048 decryption. OpenSSL uses a fixed window size $w = 5$ for exponentiation of this size. Note that RSA decryption involves another exponentiation, with d_q, and uses the Chinese Remainder Theorem to combine their results. However, the two exponentiations use exactly the same code and d_p and d_q are of the same size, so results for recovering d_q are statistically the same to those shown here for recovering d_p.

For each device, our training uses signals that correspond to 15 decryption instances, one for each of 15 randomly generated but known keys, and with ciphertext that is randomly generated for decryption. Note that these 15 decryptions provide around 12 thousand examples of S-S signal snippets, 3 thousand S-U, 3 thousand U-S, and 15 U-X snippets. This

is more than enough examples of each control flow possibility to distinguish between these control flow possibilities accurately. More importantly, this provides on average 1,500 snippet examples for each of the 100 ($2 * 5 * w$) clusters whose centroids are used as reference snippets when recovering the bits of the unknown secret exponents. Note that using larger RSA keys proportionally increases the number of snippets produced by each decryption, while w changes little or not at all. Thus for larger RSA keys we expect that even fewer decryptions would be needed for training.

After training we perform the actual attack. We randomly generate 135 RSA-2048 keys, and for each of these keys we record, demodulate, and upsample (see Section 2.4.1) the signal that corresponds to *only one decryption* with that key, using a ciphertext that is randomly generated for each decryption. Next, the signal that corresponds to each decryption is processed to extract the relevant snippets from it (see Section 2.4.2). Then, as described in Section 2.4.3, each of these snippets is matched against reference snippets (from training) to identify which of the control-flow possibilities each snippet belongs to and, for S-S and S-U snippets, which bit-position in the exponent (and the window) the snippet corresponds to. Finally, S-S and S-U snippets are matched against the 20 clusters that correspond to its position in the window to recover the value of the bit at that position in the secret exponent.

The metric we use for the success of this attack is the success rate for recovery of exponent's bits, i.e. the fraction of the exponent's bits for which the recovery produces the value that the secret exponent at that position actually had. To compute this success rate, we compare the recovered exponents to the actual exponents d_p and d_q that were used, counting the bit positions at which the two agree and, at the end, dividing that count with the total number of bits in the two exponents.

The maximum, median, and minimum success rate for each of the three targeted devices is shown in Figure 2.9. We observe that the success rate of the attack is extremely high - among all decryptions on all three devices the lowest recovery rate is 95.7% of the

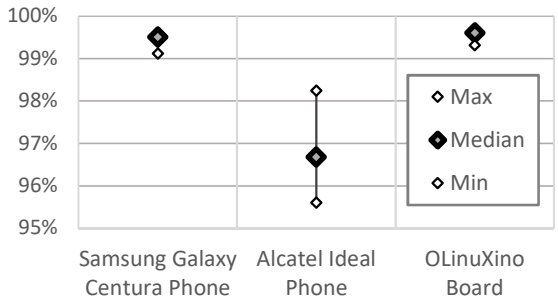

Figure 2.9: Success rate for recovery of secret exponent d_p's bits during only one instance of RSA-2048 decryption that uses that exponent. For each device, the maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown.

bits. For the OLinuXino board, most decryption instances (>85% of them) had all bits of the exponent recovered correctly, except for the most significant 4 bits. These 4 bits are processed before entering the code in Figure 2.3 to leave a whole number of 5-bit windows for that code, so we do not attempt to recover them and treat them as erasures. Among the OLinuXino decryption instances that had any other reconstruction errors, nearly all had only one additional incorrectly recovered bit (error, not erasure), and a few had two.

The results for the Samsung phone were slightly worse – in addition to the 4 most significant bits, several decryption instances had one additional bit that was left unknown (erasure) because of an interrupt that occurs between the derivative-of-moving-median peak and the end of the snippet that follows it, which either obliterates the peak or prevents the snippet from correctly being categorized according to its control flow. In addition to these unknown (but known-to-be-unknown) bits, for the Samsung phone the reconstruction also produced between 0 and 4 incorrectly recovered (error) bits.

Finally, for the Alcatel Ideal phone most instances of the encryption had between 13 and 16 unknown bits in each of the two exponents, mostly because activity on the other three cores interferes with the activity on the core doing the RSA decryption), and a similar number of incorrectly recovered bits (errors).

To examine how the results would be affected when training using signals collected on

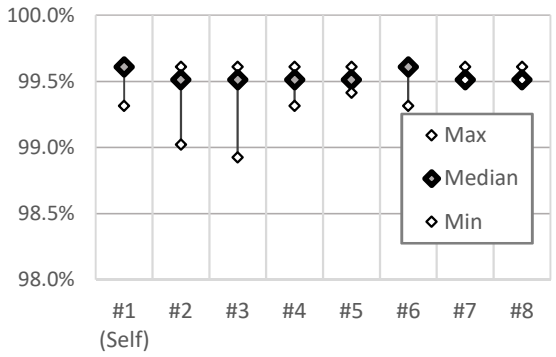

Figure 2.10: Success rate for recovery of secret exponent d_p's bits during only one instance of RSA-2048 decryption that uses that exponent, when training on OLinuXino board #1 and then using that training data for unknown exponent recovery on the same board and on seven other boards. For each device, the maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown.

one device and then recovering exponent bits using signals obtained from another device of the same kind, we use eight OLinuXino boards[1], which we label #1 through #8. Our training uses signals obtained only from board #1, and then the unknown keys are used on each of the eight boards and subjected to analysis using the same training data (from board #1). The results of this experiment are shown in Figure 2.10, where the leftmost data points correspond to training and recovery on the same device, while the remaining seven sets of data points correspond to training on one board and recovery on another.

These results indicate that training on a different device of the same kind does not substantially affect the accuracy of recovery.

Finally, for each RSA decryption instance, the recovered exponent bits, using both the recovered d_p and the recovered d_q, were supplied to our implementation of the full-key reconstruction algorithm. For each instance, the correct full RSA private key was reconstructed within one second on the Core i7-based Surface Pro 4 tablet, including the time needed to find the k_p and k_q coefficients that were not known a priori. This is an expected result, given that even the worst bit recovery rates (for the Alcatel phone) correspond to a an error rate of about 1.5%, combined with an erasure rate of typically 1.5% but sometimes as

[1]The OLinuXino boards are much less expensive than the phones, so we could easily obtain a number of OLinuXino boards

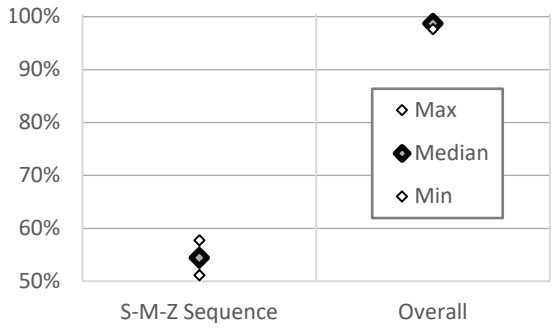

Figure 2.11: Success rate for recovery of secret exponent d_p's bits during only one instance of RSA-2048 decryption that uses that exponent for sliding-window exponentiation. The maximum, median, and minimum success rate among decryption instances (each with a different randomly generated key) is shown for recovery that only uses the snippet-type sequence (S-M-Z Sequence), and for recovery that also recovers window bits from U-S and Z-S snippets (Overall).

high as 3% (depending on how much system activity occurs while RSA encryption is execution on the phone), which is well withing the range for which our full-key reconstruction is extremely efficient.

2.5.4 Results for the Sliding-Window Implementation

To improve our understanding of the implications for this new attack approach, we also apply it to RSA-2048 whose implementation uses OpenSSL's sliding-window exponentiation – recall that this was the default implementation used in OpenSSL until it switched to a fixed-window implementation in response to attacks that exploit sliding-window's exponent-dependent square-multiply sequence.

In these experiments we use 160 MHz of bandwidth and target the OLinuXino board. Recall that in a sliding-window implementation our method can categorize the snippets according to their beginning/ending point to recover the sequence of zero-squaring (Z), window-squaring (S), and result update (M) occurrences. The fraction of the exponent's bits recovered by this sequence reconstruction (shown as "S-M-Z Sequence" in Figure 2.11) is in our experiments between 51.2% and 57.7% with a median of 54.5%. This sequence-based recovery has produces no errors in most cases (keys), and among the few encryptions

that had any errors, none had more than one.

In our attack approach, after this sequence-based reconstruction, the U-S and Z-S snippets are subjected to further analysis to recover the remaining bits of the window computed in each U-S and Z-S snippet. At the end of this analysis, the fraction of the exponent's bits that are correctly recovered ("Overall" in Figure 2.11) is between 97.7% and 99.6%, with a median of 98.7%.

This rate of recovery for exponent bits provides for very rapid reconstruction of the full RSA key. However, we note that it is somewhat inferior to our results on fixed-window exponentiation on the same device (OLinuXino board), in spite of using more bandwidth for attacks on sliding-window (160MHz bandwidth) than on fixed-window (40MHz bandwidth) implementation. The primary reason for this is that in the fixed-window implementation each analyzed snippet corresponds to examining only one bit of the exponent, whereas in the sliding-window implementation $wmax = 6$ bits of the exponent are examined in a single U-S or Z-S snippet, while the exponent-dependent variation in the snippet is not much larger. Since sliding-window recovery tries to extract several times more information from about the same amount of signal change, its recovery is more affected by noise and thus slightly less accurate.

2.6 Mitigation

We focus our mitigation efforts on the fixed-window implementation, which is the implementation of choice in the current version of OpenSSL, and which already mitigates the problem of exponent-dependent square-multiply sequences and timing variation. We identify three key enablers for this attack approach, which roughly correspond to discussion in Sections 2.4.1, 2.4.2, and 2.4.3. Successful mitigation requires removing at least one of these enablers, so we now discuss each of the attack enablers along with potential mitigation approaches focused on that enabler.

The first enabler of the specific attack demonstrated in this chapter is the existence

of computational-activity-modulated EM signals around the processor's clock frequency, and the attacker's ability to obtain these signals with sufficient bandwidth and signal-to-noise ratio. Potential mitigation thus include circuit-level approaches that reduce the effect the differences in computation have the signal, additional shielding that attenuates these signals to reduce their signal-to-noise ratio outside the device, deliberate creation of RF noise and/or interference that also reduces the signal-to-noise ratio, etc. We do not focus on these mitigation because all of them increase the device's overall cost, weight, and/or power consumption, all of them are difficult to apply to devices that are already in use, and all of them may not provide protection against attacks that use this attack approach but through a different physical side channel (e.g. power).

The second enabler of our attack approach is the attacker's ability to precisely locate, in the overall signal during an exponentiation operation, those brief snippets of signal that correspond to examining the bits of the exponent and constructing the value of the window. A simple mitigation approach would thus insert random additional amounts of computation before, during, and/or after window computation. However, additional computation that has significant variation in duration would also have a significant mean of that duration, i.e. it would slow down the window computation. Furthermore, it is possible (and indeed likely) that our attack can be adapted to identify and ignore the signal that corresponds to this additional activity.

The final (third) enabler of our attack approach is the attacker's ability to distinguish between the signals whose computation has the same control flow but uses different values for a bit in the exponent. In this regard, the attack benefits significantly from 1) the limited space of possibilities for value returned by `BN_is_bit_set` – there are only two possibilities: 0 or 1, and from 2) the fact that the computation that considers each such bit is surrounded by computation that operates on highly predictable values – this causes any signal variation caused by the return value of `BN_is_bit_set` to stand out in a signal that otherwise exhibits very little variation.

Based on these observations, our mitigation relies on obtaining all the bits that belong to one window at once, rather than extracting the bits one at a time. We accomplish this by using the `bn_get_bits` function (defined in `bn_exp.c` in OpenSSL's source code), which uses shifts and masking to extract and return a BN_ULONG-sized group of bits aligned to the requested bit-position – in our case, the LSB of the window. The BN_ULONG is typically 32 or 64 bits in size, so there are billions of possibilities for the value it returns, while the total execution time of `bn_get_bits` is only slightly more than the time that was needed to append a single bit to the window (call to `BN_is_bit_set` shifting the $wval$, and or-ing to update wval with the new bit). For the attacker, this means that there are now billions of possibilities for the value to be extracted from the signal, while the number of signal samples available for this recovery is similar to what was originally used for making a binary (single-bit) decision. Intuitively, the signal still contains the same amount of information as the signal from which one bit used to be recovered, but the attacker must now attempt to extract tens of bits from that signal.

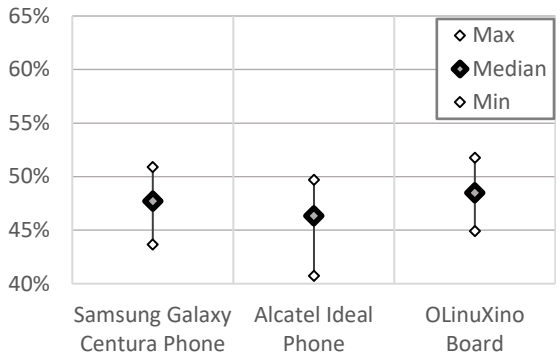

Figure 2.12: Success rate for recovery of secret exponent d_p's bits after the initial implementation of our window value randomization mitigation is applied.

This mitigation results in a slight *improvement* in execution time of the exponentiation and, as shown in Figure 2.12, with the mitigation the recovery rate for the exponent's bits is no better than randomly guessing each bit (50% recovery rate). In fact, the recovery rate with the mitigation is lower than 50% because, as in our pre-mitigation results, the bits whose signal snippets could not be located are counted as incorrectly recovered. However,

these bits can be treated as erasures, i.e. for each such bit the attacker knows that the value of the bit is unknown, as opposed to a bits whose value is incorrect but the attacker has no a-priori knowledge of that, so our recovery rate can be trivially improved by randomly guessing (with 50% accuracy) the value of each erasure, rather than having 0% accuracy on them. With this, the post-mitigation recovery rate indeed becomes centered around 50%, i.e. equivalent to random guessing for all of the bits.

This mitigation has been submitted to OpenSSL and was merged into its master source code branch on May 20th, prior to the publication of this chapter.

2.7 Conclusions

This chapter presents the first side channel attack approach that, without relying on the cache organization and/or timing, retrieves the secret exponent from a single decryption on arbitrary ciphertext in a modern (current version of OpenSSL) fixed-window constant-time implementation of RSA. Specifically, the attack recovers the exponent's bits during modular exponentiation from analog signals that are unintentionally produced by the processor as it executes the constant-time code that constructs the value of each "window" in the exponent, rather than the signals that correspond to squaring/multiplication operations and/or cache behavior during multiplicand table lookup operations. The approach is demonstrated using electromagnetic (EM) emanations on two mobile phones and an embedded system, and after only one decryption in a fixed-window RSA implementation it recovers enough bits of the secret exponents to enable very efficient (within seconds) reconstruction of the full private RSA key.

Since the value of the ciphertext is irrelevant to our attack, the attack succeeds even when the ciphertext is unknown and/or when message randomization (blinding) is used. Our evaluation uses signals obtained by demodulating the signal from a relatively narrow band (40 MHz) around the processor's clock frequency (around 1GHz), which is within the capabilities of compact sub-$1,000 software-defined radio (SDR) receivers.

CHAPTER 3

NONCE@ONCE: A SINGLE-TRACE EM SIDE CHANNEL ATTACK ON ECDSA IN GNUPG/LIBGCRYPT AND OPENSSL

3.1 Abstract

This chapter presents the first side-channel attack that recovers the secret ephemeral scalar (nonce) used in the elliptic curve point multiplication by a scalar in ECDSA, using the signal that corresponds to a single signing operation, in current versions of `GnuPG/libgcrypt` and tt OpenSSL. Specifically, these ECDSA implementations use a conditional swap to avoid creating control-flow and memory-access-pattern differences depending on the bits of the nonce, because the knowledge of the nonce allows straightforward recovery of the private key that was used in the ECDSA signing operation, and our attack uses the signal differences created by systematic differences in operand values during a conditional swap operation itself to recover each bit of the nonce.

Our experimental setup uses a custom-designed electric field probe (<$30) and a software-defined radio (SDR) receiver (<$770) to collect the electromagnetic (EM) emanations created by the victim device's processor. We deploy the attack against two Android-based mobile phones and against a Linux-based IoT development board, repeating the attack 100 times (each time with a different ECDSA key) on each device for each of the two target cryptographic packages (`libgcrypt` and `OpenSSL`), and in each of these (hundreds of) instances of ECDSA signing, the attack has successfully recovered the full ECDSA key within seconds.

Finally, we propose a mitigation based on randomizing the exclusive-or mask in the conditional swap operation, which avoids creating a systematic condition-dependent difference in operand values for exclusive-or operations during the conditional swap. We

have confirmed that this mitigation is effective in preventing this and similar attacks, and are currently working on submitting this mitigation to both `GnuPG` and `OpenSSL`, and expect the mitigation to be merged into both source code repositories prior to publication of this work.

3.2 Motivation

Physical side channel attacks exploit the physical side-effects of computation to extract sensitive information, such as cryptographic keys, that is used during that computation. The physical side-channel signals include electromagnetic emanations, which are created by changes in the flows of current within the computational device [56, 57, 58, 59, 60, 61], variations in power consumption [26, 62, 63, 64], acoustic emanations [65, 30], and temperature variation [66]. Most prior physical side-channel attacks exploit the large differences in signals that are created when program execution takes different paths depending on a condition that reveals information about the secret key. For example, some attacks exploit the changes to the sequence of large-number `square` and `multiply` during modular exponentiation in RSA [67, 68], ElGamal [69] and DSA [70] implementations. Analogously, changes to the sequence of elliptic curve (EC) point `double` and `add` operations, as well as other control flow that leaks information about the scalar during the EC point multiplication by a scalar, has been exploited to attack prior implementations of EC-based cryptographic algorithms [71, 72, 73, 74]. Other (non-physical) attacks on earlier implementations of ECDSA include exploiting secret-dependent differences in cache behavior (memory access patterns) [75, 76, 77, 78]. Finally, in some attacks the message of the base EC point is chosen by the attacker [79] in a way that produces values with distinguishable signals in the EC point `double` and `add` operations during EC point-scalar multiplication.

To mitigate these side-channel attacks, in recent versions of cryptographic packages, such as `GnuPG/libgcrypt` and `OpenSSL`, point and message blinding is applied prior to performing point-scalar multiplication, to prevent chosen-message and chosen-base-

point attacks. The point-scalar multiplication itself is implemented such that the execution time, control-flow, and memory access pattern, are all independent of the value of the (secret) scalar. To produce the correct result without introducing timing, control-flow, and access-pattern variation, these implementations rely on a conditional swap operation, in two EC points are either swapped or not, depending on the value (0 or 1) of a *swap condition*, using a sequence of bit-masking and exclusive-or operations whose sequence of executed instruction, and sequence of accessed data addresses, remains the same regardless of the value of the swap condition. This approach is needed to avoid leaking the value of the swap condition, which has a direct relationship to the bits of the secret nonce, through various side channels. Overall, the current implementations of point-scalar multiplication used by ECDSA in `libgcrypt` and `OpenSSL` are designed to mitigate existing physical and cache-based side channel attacks.

3.2.1 Our Contributions

This chapter presents a side-channel attack that recovers the secret ECDSA key by analyzing the signal that corresponds to a single ECDSA signing operation in the current (as of this writing) versions of `libgcrypt` and `OpenSSL`. Specifically, our attack 1) identifies the signal snippet that corresponds to each instance of the conditional swap operation, 2) determines which of the two possible values of the swap condition each signal snippet corresponds to, 3) uses the values of the swap conditions, which directly correspond to individual bits of the secret nonce in an ECDSA signing operation, to construct the set of possible nonce values, 4) reconstructs the full private/public ECDSA key pair that corresponds to each nonce candidate and 5) identifies which of the key pair candidates matches the (known) public ECDSA key of the signing entity, and, thus, contains the signing entity's actual private ECDSA key.

We experimentally evaluate this attack for both `libgcrypt` and `OpenSSL`, for three different target devices (two Android-based mobile phones and an IoT development board),

using as training only two signal snippets, one for a point-scalar multiplication step where the swap condition was 0, and one where the swap condition was 1. These training signals were collected by executing an EC point-scalar multiplication, with a known value of the nonce, on another physical instance of the target device. We repeat the attack 100 times for each software-device combination, using a single signing operation, with a newly generated ECDSA key, in each of these 600 (100 x 2 software packages x 3 devices) attacks instances. We find that the, in all 600 of these attack instances, the full private ECDSA key is successfully recovered, with no attack instance requiring more than 1.3s of single-core processing time for analysis and key reconstruction.

The key insight on which the attack is based is that, in both `libgcrypt` and `OpenSSL`, within the conditional swap operation, the values of operands in exclusive-or operations are systematically biased depending on the value of the condition. Specifically, when the swap condition is zero (no actual swap is needed), exclusive-or operations all have zero-valued operands, whereas when the swap condition is one (actual swap is needed) the exclusive-or operations have operands whose bits are unbiased (1-valued bits appear just as often as 0-valued bits do). Because each conditional swap operations performs tens of exclusive-or operations, all with the same bias in their operands, the resulting overall difference in the side-channel signal is many times larger than the typical difference caused by bit-level variation in operands of a single instruction. Therefore, even when the noise in the side channel signal is significantly larger than the signal variation caused by different values of individual instruction operands, the systematic difference caused by many instructions, all having the same bias in all bits in one of their operands, becomes detectable in the signal.

Finally, we leverage this insight to develop a mitigating software change for both `libgcrypt` and `OpenSSL`, which prevents this style of attack with a negligible impact on performance. The mitigation consists of randomizing the exclusive-or difference values in the conditional swap operation, which results in all exclusive-or operations, regardless of the swap condition, being performed with operands whose bits are unbiased. We have

confirmed that this mitigation is effective in preventing this and similar attacks. We are currently preparing to submit this mitigation to `GnuPG` and `OpenSSL`, and will work to ensure that mitigation is merged into the main trunk of both source code repositories prior to publication of this work.

3.2.2 Threat Model

We assume that the attacker can place the probe, along with a compact receiver, into close physical proximity to the device that is performing an ECDSA signing operation. For example, a smart-infrastructure or smart-city device may be in a physical location, and may use Elliptic Curve (EC) cryptography to establish secure connections and/or sign messages it transmits over the Internet. Another example would when an adversary hides the probe and the receiver in a location where mobile phones and computer systems may be used, e.g. under a desk, table, of charging station at an airport, in a coffee shop, etc.

We also do *not* assume that the adversary can choose (or even know) the message to which ECDSA signing operation will be applied, or even choose the curve and/or the base EC point for ECDSA signing operations, and we assume that the ECDSA implementation under attack *does* utilize message and base-point blinding to mitigate such chosen-message and chosen-point attacks.

Finally, we assume that it is highly advantageous for the attacker to recover the private key when it is used, without having to wait for another use of the same key. This assumption is often realistic in practice, because in many scenarios (e.g. smart-infrastructure devices) the attacker can only briefly place the probe in close proximity to the victim system without attracting suspicion, while in other scenario (coffee shop, airport) the victim system is placed close to the attacker's probe for only a short time. Furthermore, the victim system may perform private-key EC operations sporadically, e.g. to set up a connection or to sign emails that take a while to compose, so the attacker may not be able to collect signals for more than one, or perhaps a few, signing operations.

3.2.3 Targeted Software and Hardware

The software we target are the latest stable released versions of `GnuPG/libgcrypt` (version 1.8.4) and `OpenSSL` (version 1.1.1a). In the target version of `libgcrypt`, ECDSA uses EC point `add` and `double` operations with a conditional swap to implement constant-time, constant-control-flow, constant-data-access-pattern EC point multiplication by a scalar. Notably, the `ECDH` implementation in the same version of `libgcrypt` uses a the constant-time `Montgomery` ladder and conditional swaps for the same purpose. In `OpenSSL`, ECDSA uses constant-time `Montgomery` ladder and conditional swaps. Since our attack actually targets the conditional swap operation, it can target all of these point-scalar multiplication implementations, regardless of whether they use separate `add` and `double` operations or a `Montgomery` ladder step. The curves we use in our experiments are curve `Ed25519` for `GnuPG` and the `secp256k1` curve for `OpenSSL`. However, ECDSA signature computation, regardless of curve equations and choice of a base point, have no material impact on our attack as they use the same point-scalar multiplication, and thus the same conditional swap, program code. Finally, both `libgcrypt` and `OpenSSL` perform blinding of point and/or message values, but that has no material effect on our attack because the value of the condition remains the same in each instance of the conditional swap, regardless or message/point blinding.

The hardware we target are two ARM-based mobile phones (Alcatel Ideal and ZTE ZFIVE), running under different versions of the Android operating system, and also an ARM-based IoT prototype board (A13-OLinuXino), running under Debian Linux. We consider our attack to be a non-intrusive but close-proximity attack, as the probe in our experiments is placed in close proximity to each target device, but without opening the device's enclosure and without direct physical contact with the device.

3.3 Background

3.3.1 Overview of ECDSA

ECDSA uses an Elliptic Curve (EC) to compute a cryptographic signature using a private key and to verify the signature using the corresponding public key. First, the participating parties must agree on a choice of curve parameters: the elliptic curve field and equation, denoted here as CURVE, the base point of prime order on the curve, denoted here as G, and an integer order of G, denoted here as n. Given the curve parameters (CURVE,G,n), the signing entity (which we refer to as Alice) secretly chooses as its private key a random positive integer d_A such that $0 < d_A < n$, and computes the curve point $Q_A = d_A \times G$ that will serve as the public key that corresponds to the secret key d_A. The public key is then provided to the verifying entity, which we refer to here as Bob.

To sign a message, Alice first computes e, the cryptographic hash of the message, and then computes z, the value of e truncated to the bit-length of n. Next, Alice randomly chooses an ephemeral secret positive integer (nonce) k such that $0 < k < n$, calculates the curve point $(x, y) = k \times G$ where $\times$ stands for an EC point multiplication by a scalar, then computes $r = x \ mod \ n$ and $s = k^{-1}\dot{(}z + r\dot{d}_A) \ mod \ n$. If either r or s is zero, a new k is chosen and the signature is recomputed, otherwise the signature consists of the pair (r, s).

To verify the signature, Bob also computes the cryptographic hash of the message and the corresponding value of z. Bob then computes $w = s^{-1} \ mod \ n$, $u_1 = z\dot{w} \ mod \ n$, and $u_2 = r\dot{w} \ mod \ n$, then computes the curve point $(x, y) = u_1 \times G + u_2 \times Q$. The signature is valid when r and s are both within the interval $[1, n-1]$, $(x, y)! = O$, and $r \equiv x \ (mod \ n)$.

The attacker (which we refer to as Eve) already knows the curve parameters (CURVE,G,n), and can typically obtain Alice's public key Q_A and a large number of messages signed by Alice. Under these conditions, Eve's attempt to calculate the private key d_A requires solving a complex discrete logarithm problem. However, if Eve can obtain the secret nonce k for even one signature, the secret key d_A can be trivially recovered as $d_A = (s\dot{k} - z)/r$.

One way for Eve to obtain the value of k is to mount a side channel attack during Alice's computation of the signature. Since the value of k is ephemeral - it is generated, used to compute a signature for one message, and then discarded, Eve can try to extract the value of k when it is used in EC point multiplication ($k \times G$), or when its inverse is used as a multiplicand to compute s. Of the two, the point multiplication is typically the more promising target for the side channel attacks because it is more time-consuming and because its implementation performs operations on curve points depending on the bits of k, which has a tendency to leak information about k unless the implementation is very carefully constructed to avoid doing so.

3.3.2 Point Multiplication by a Scalar

The naive implementation of EC point multiplication is shown in Figure 3.1.

```
1  EC_point_zero(r); // r=0
2  // For each bit of the scalar k
3  for(b=bits-1;b>=0;b--){
4     EC_point_double(r,r); // r=2*r
5     if(BN_is_bit_set(k,b))
6        EC_point_add(r,r,p); // r = r+p
7
8  }
```

Figure 3.1: A naive double-and-add implementation of EC point multiplication by a scalar.

In this implementation of point multiplication, for each bit of the scalar k, the previous result r is doubled and, if the bit in k is non-zero, the point p is added to that result. The point-double and point-add operations differ in both the code that is executed and the data that is accessed, so various side channels can be used to recover the sequence of these operations and, from that sequence, recover the bits of the scalar k. For example, instruction cache accesses can be used to determine when each point-double and point-add function call occurs, the timing of data cache accesses to p can be used to determine which point-double operations are followed by point-add operations (note that p is only accessed

during a point-add), and various analog side channel signals can be used to determine when each point-double and point-add is executed.

For performance reasons, instead of a binary double-and-add implementation, both `libgcrypt` and `OpenSSL` have until recently used an implementation based on the non-adjacent form (NAF) of the scalar. In the NAF representation the scalar is still represented as a sequence of digits, but each digit now represents multiple bits of the scalar. After pre-computing the value of the point multiplied by each of the possible value of a digit, the NAF-based implementation requires only one point-add for each non-zero digit in the NAF representation of the scalar, and this significantly reduces the number of point-add operations that are needed for the overall point multiplication. However, the pattern of accesses to the table of pre-computed point values now directly corresponds to the NAF representation of k, which allows cache-based attacks to easily recover k [75, 77, 78, 76]. Furthermore, the point-add is still skipped for a zero-valued NAF digit in k, so the sequence of point-double and point-add operations still leaks partial information about k. This has been exploited by analog side channel attacks, where partial information from multiple signing operations (that use the same private key d_A, but different scalars k) was combined to eventually recover d_A [72].

```
1  EC_point_zero(r); // r=0
2  // For each bit of the scalar k
3  for(b=bits-1;b>=0;b--){
4    EC_point_double(r,r);     // r=2*r
5    EC_point_add(tmp,r,p);    // tmp=r+p
6    // Swap r and tmp if b-th bit of k is 1
7    curr_bit=BN_is_bit_set(k,b);
8    EC_point_swap_cond(r,tmp,curr_bit);
9  }
```

Figure 3.2: `libgcrypt`'s approach to constant-time point multiplication.

To mitigate the side-channel problems of the NAF-based implementation, both `libgcrypt` and `OpenSSL` have recently switched to using constant-time implementations of point multiplication for during ECDSA signing. Figure 3.2 shows the constant-time implementa-

tion adapted from `libgcrypt`'s source code (function `_gcry_mpi_ec_mul_point()` in `mpi/ec.c`). This is a double-and-add implementation but, unlike the naive implementation, it executes a point-add for every bit in k, so the sequence of point-double and point-add operations (and the data access pattern) no longer leaks information about k. The output of the point-add operation (tmp in our code example) is then conditionally swapped with the result r, using the bit of k as a condition that dictates whether the swap occurs or not. When a zero-valued bit is encountered in k, tmp and r are left unchanged, which effectively discards the result of the point-add operation. Conversely, when a one-valued bit is encountered in k, the swap results in using the output of the point-add operation as the new value of r.

```
1  EC_point_ladder_prep(r,s,p);
2  prev_bit=1;
3  // For each bit of the scalar k
4  for(b=bits-1;b>=0;b--){
5     curr_bit=BN_is_bit_set(k,b);
6     EC_point_swap_cond(r,s,curr_bit^prev_bit);
7     EC_point_ladder_step(r,s,p);
8     prev_bit=curr_bit;
9  }
10 EC_point_swap_cond(r,s,prev_bit);
11 EC_point_ladder_post(r,s,p);
```

Figure 3.3: `OpenSSL`'s approach to constant-time point multiplication.

Figure 3.3 shows the constant-time implementation adapted from `OpenSSL`'s source code (function `ec_scalar_mul_ladder()` in `crypto/ec/ec_mult.c`). It differs from `libgcrypt`'s implementation in that the doubling and add operations are integrated into a single Montgomery ladder [80, 81, 82] step, which also changes the condition used for the swap - rather than performing the swap depending on each individual bit of the scalar k, the swap is now performed when the i-th bit of k differs from the previous one.

In both `libgcrypt`'s and `OpenSSL`'s implementation of EC point multiplication, the conditional swap is used to avoid having the control flow of the point multiplication depend on individual bits of the scalar k. The implementations of the conditional swap itself in

```
1  EC_point_swap_cond(a,b,cond){
2    // When cond is 0, set mask to all-zeros
3    // When cond is 1, set mask to all-ones
4    mask=0-cond;
5    // For each machine word in the
6    // EC point representation
7    for(i=0;i<nwords;i++){
8      delta = (a->w[i] ^ b->w[i]) & mask;
9      a->w[i] = a->w[i] ^ delta;
10     b->w[i] = b->w[i] ^ delta;
11   }
12 }
```

Figure 3.4: Constant-time conditional `swap`. When cond is true, EC points a and b are swapped, otherwise they are left unchanged.

`libgcrypt` and in `OpenSSL` are very similar, and Figure 3.4 shows the implementation of conditional swap adapted from `libgcrypt`'s function `_grcy_mpi_swap_cond`) in source code file `mpi/mpiutil.c`. An EC point is stored as one large numbers for each coordinate, and each number is stored as a sequence of machine words. For each machine word in the EC point representation, a bitwise exclusive-or (XOR) is used to compute the bit-wise difference between the word in EC point a and the corresponding word in EC point b. The $mask$ is then applied to this difference, such that the difference is either kept as-is or zeroed out, depending on the condition $cond$. This masked difference is then applied (via XOR operations) to the two words. The ends result is that, when $cond$ is true, the values of the two EC points are swapped, but when $cond$ is false the two points are left unchanged. The key property of this conditional swap is that the same instruction sequence is executed, and the same sequence of data accesses is performed, regardless of the value of the condition, which prevents cache-based and many analog-signal side channel attacks from obtaining information about the value of the swap's condition, and thus about the bits of the scalar k in the EC point multiplication.

Because constant-time point multiplication implementations in both `libgcrypt` and `OpenSSL` result in executing exactly the same sequence of instructions, and exactly the

same sequence of data accesses, regardless of the value of scalar k, they prevent side channel attacks that exploit either instruction or data cache behavior, as well analog side channel attacks that rely on detecting signal differences caused by executing different program code depending on the bits of the scalar k.

However, we make a key observation that the conditional swap implementation creates systematic condition-dependent differences in the values of operands used by XOR instructions, and in the bit-toggling activity among words in the internal representation of the two EC points. Specifically, when $cond$ is true, the $mask$ is all-ones, which results in values of $delta$ that have about the same number of 0-valued and 1-valued bits, and this results in toggling about half of the bits in each word of EC points a and b. In contrast, when $cond$ is false, the $mask$ is all-zeroes, so the $delta$ for each pair of words is zero, and no bits are toggled in any of the words that represent EC points a and b. The standard curve parameters we use result in a 256-bit number for each coordinate of an EC point, which means that operations on several tens of 32-bit machine words are systematically affected this way by the value of of $cond$, and we exploit the resulting difference in the processor's electromagnetic emanations to reconstruct the secret nonce k, and thus the private key, using the signal collected during only one instance of a signing operation.

3.4 The Nonce@Once Attack

This section describes our new attack on the constant-time point multiplication implementations in `libgcrypt` and `OpenSSL`. In this attack, the processor's electromagnetic (EM) emanations are recorded during execution of a single instance of an ECDSA signing operation. This recorded EM signal is then analyzed to identify the parts of the signal that correspond to each conditional swap operation, and for each of these operations to identify the value of the swap condition (true of false). The sequence of swap condition values is then used to reconstruct the ephemeral secret nonce k used in the ECDSA signing operation and, by combining the recovered value of k with the (non-secret) ECDSA signature,

recover the signing party's private ECDSA key.

This attack is the first side channel attack that targets the constant-time implementations of point multiplication that have been recently introduced in `libgcrypt` and `OpenSSL`, which were both designed to resist side channel attacks. This attack is very powerful in that it efficiently recovers the secret ECDSA key after observing only one instance of an ECDSA signing operation.

3.4.1 EM Signal Acquisition

The side channel signal used in our attack consists of electromagnetic (EM) emanations created by the victim system's processor as it executed instructions. Although the actual values of the bits in each of the operands in an individual instruction create only minuscule differences in this emanated EM signal, the values of operands during a conditional swap operation create thousands of such bit-wise differences in a systematic way, making the resulting difference in EM emanations strong enough to be observed when the signal is collected using our custom-made high-gain probes that are placed just outside be victim system's (unopened) case. The signal from the probe is then filtered, down-converted, and digitally recorded using an off-the-shelf software-defined radio (SDR) receiver, such that the recorded signal samples correspond to several megahertz (MHz) of radio-frequency bandwidth around the victim system's processor clock frequency. The recorded signal is then digitally filtered and demodulated before it is subjected to the custom signal analysis that implements our attack.

3.4.2 Identifying Signal Snippets that correspond to Conditional Swap Operations

The information we extract from the signal is the condition (swap or not) used in each conditional-swap operation withing the EC point-scalar multiplication. The first step in the signal analysis is to identify the part of the signal that corresponds to the overall point-by-scalar multiplication. This can be trivially accomplished by changing the source code of

`libgcrypt` and `OpenSSL` to create a highly recognizable signal pattern just before and just after the point-by-scalar multiplication function is called during an ECDSA signing operation, or to record a high-resolution time-stamp and identify the corresponding real-time point in the signal. However, in most realistic attack scenarios such modification of the victim's code would not be possible, so instead we execute the victim code as-is, and identify the point-scalar multiplication purely through signal analysis.

The key observation for this signal analysis is that most of the execution time in point-scalar multiplication is spent on repeated point-double and point-add operations in `libgcrypt`, and on point-ladder-step operations in `OpenSSL`, and these point operations are designed such that all instances of their execution have as little variation as possible in their execution time, control flow path, and data access pattern. If a training sample of a point-double and point-add (for `libgcrypt`), or a training sample of a point-ladder-step (for `OpenSSL`) is available, we use moving correlation with that sample to identify the part of the signal that contains an appropriate number of repetitions of the sample. In our experiments we find that the point-double, point-add, and point-ladder-step operations are long enough, and have enough prominent signal features, to only require one training sample of each, i.e. the training signal corresponds to the work for only one bit in the scalar k during one signing operation, and we do not need the knowledge of the value of that bit in k.

Since the point operation always follows the same control flow path, the duration of the period can be estimated, either by statically analyzing the program code, or by running the point-by-scalar multiplication code on *another* computer system to obtain an estimate (within one or two orders of magnitude) for the number of processor cycles for a point-multiplication step. Since this estimate need not be very precise, the profiling can be performed on a very different computer system, e.g. an x86-based PC workstation with 2.7 GHz clock frequency provides a sufficiently accurate estimate to identify the point-scalar multiplication when attacking an ARM-based mobile phone with an 800MHz clock

frequency. The number of periods corresponds to the number of bits in k, and it is determined by the curve parameters used in ECDSA. In `OpenSSL` the number of steps is always equal to the maximum number of bits in the nonce k, which is determined by the curve parameters, and is equal to 256 bits or the `ecp256k1` curve we use in OpenSSL. In `libgcrypt` we used `Ed25519`, which also results in the maximum nonce size of 256 bits, but in `libgcrypt` the point-scalar multiplication omits the steps that correspond to leading zeros in the nonce, so the actual number of steps can be smaller than 256. However, very small nonces (many leading zeros) are highly unlikely, so if we require the number of steps to be between 128 and 256, there is only an astronomically small chance (about one occurrence in three hundred million decillion signing operation) of a false negative (missing a point-scalar multiplication in the overall signal). In our experiments we have experienced no false positives (falsely identifying a point-scalar multiplication), but if such a false positive were to occur, our reconstruction of the private ECDSA key would find that no private-key candidate constructed using this signal is viable (does not match the signer's known public key).

The next step in our analysis is to identify the parts of the signal that corresponds to individual conditional swap operations, so that each snippet can be analyzed to identify the value of the condition (swap or no swap) that was used in that instance of the conditional swap. The conditional swap itself is too brief to be reliably detected by matching to a training sample of the swap itself, but we can leverage the observation that the conditional swap is executed between a point-add in one step and the point-double in the next step (in `gnupg`) and between one point-ladder-step and the next one (in `OpenSSL`). Therefore, if a training sample of each of these point operations is available, we can match them in the signal, identify where each such operation begins and ends, and then extract the signal between the end of a point-add (or the end of a point-ladder-step) and the beginning of the next point-double (or the next point-ladder step) as the swap-related signal snippet.

Even though `libgcrypt` and `OpenSSL` have similar overall approaches to imple-

menting the overall point-scalar multiplication and, especially, the conditional swap, their actual program code differs enough to prevent using the actual signal from one to attack the other. To illustrate the similarities and differences between their signals, Figure 3.5 shows the signal that corresponds to a conditional swap and the next point operation in `libgcrypt` and in `OpenSSL`.

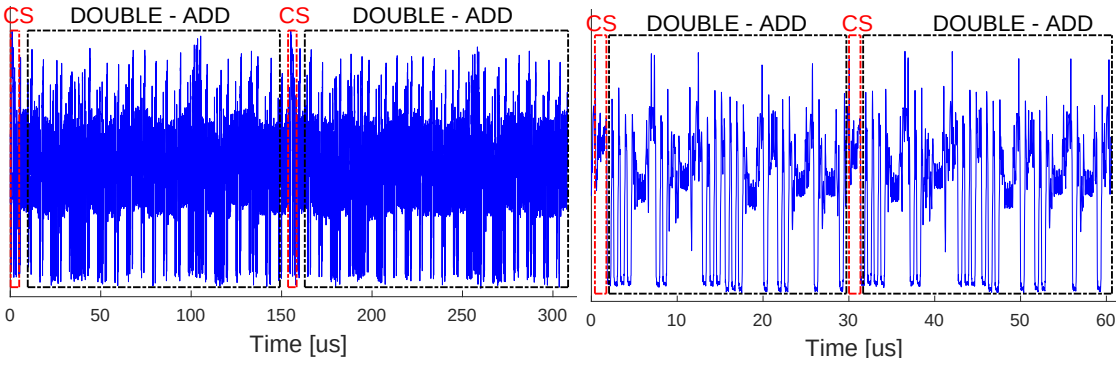

Figure 3.5: Signal example for `libgcrypt` (left) and for `OpenSSL` (right). The signal for a conditional swap is indicated by a solid red rectangle, while the signal for the next point operation is indicated by a dashed black rectangle.

3.4.3 Recovering the Value of the Swap Condition from a Snippet

Once the signal snippet for each instance of the conditional swap is identified, the final step in our signal analysis is to categorize them according to the value of the condition that has been used. As described in Section 3.3, the conditional swap constructs a mask, which is either all-zeros or all-ones depending on the swap condition, and then applies this mask to an exclusive-or-based swap for each machine word in the internal representation of two EC points, such that the two words remain the same if the swap condition was false, or the value of the two words are exchanged it the swap condition was false. The internal representation of an EC point includes the large-number coordinates of the point, where each coordinate is stored as an array of several machine words, as well as several machine words that contain metadata about the EC point, such as its size (in machine words), sign, and other information. The default curve parameters we use in our experiments result in

256-bit coordinates, i.e. eight 32-numbers per coordinate, with three dimensions, for a total of 24 words for coordinates, and nearly 30 words total per point. Thus a conditional swap whose condition is false performs about 60 exclusive-or operations whose one operand is always zero, and accesses about 60 machine words that are all left unchanged, while a conditional swap whose condition is true performs those 60 exclusive-or operations with non-zero operands that have about the same number zeros and ones, and changes about 60 machine words by toggling about half of the bits in each word. The signal snippets that correspond to `libgcrypt`'s conditional swap with one-valued and zero-valued condition are shown in the top part of Figure 3.6, where the signal differences produced by different values of the swap condition can be clearly observed, and we can even discern three bursts of these signal differences that correspond to the three calls to `libgcrypt`'s internal `_gcry_mpi_swap_cond` function, once for each coordinate, during the execution of the `point_swap_cond` function.

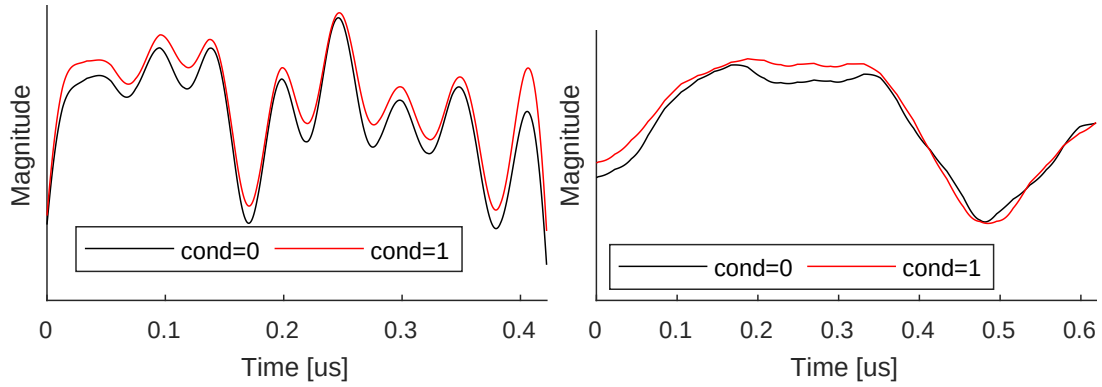

Figure 3.6: Signal snippets that correspond to conditional swap in `libgcrypt` (left) and `OpenSSL` (right) when the swap condition is true (points are actually swapped, signal shown in blue) and when the swap condition is false (points unchanged, signal shown in red).

Because the signal snippets exhibit significant differences depending on the swap condition, when training examples of true-condition and false-condition snippets are available, we use simple correlation between the candidate snippet and two training examples of each kind (two for false-condition and two for true-condition), and we assign the swap condi-

tion of the best-matching (highest-correlation) training example as the candidate snippet's reconstructed value of the swap condition.

3.4.4 Candidate Nonce Values

From the signal analysis, we obtain the reconstructed value for the swap's condition (variable $cond$ in Figure 3.4) from each swap-related signal snippet that has been identified, and from the timing between identified signal snippets we obtain the positions of the missing swap-related snippets. In `libgcrypt` the reconstructed values for $cond$ directly correspond to individual bits of the nonce k, while the missing snippets correspond to unknown bits in the reconstructed nonce. We thus construct a set of candidate nonce values where all possible values for the unknown bits are represented, and we subject this set of candidate nonces to our algorithm that recovers ECDSA private keys (Section 3.4.5). Note that each missing snippet doubles the number of candidate nonces, so recovery of private keys can only be efficient when relatively few snippets are missing.

For `OpenSSL`, the reconstructed value of $cond$ in each swap-related signal snippet corresponds to the difference in value between the current bit in k and the previous one. Therefore, a missing snippet results in having to consider both possibilities for the current bit of k, and also having to perform reconstruction of the rest of the nonce under each of the two assumptions. This is only a minor complication, because a change in the value of one bit in the nonce can be accounted for by toggling all bits that are less significant. Aside from that, the effect of missing snippets on the number of candidate nonces is the same as in `libgcrypt` – each missing snippet doubles the number of candidate nonces. Additionally, `OpenSSL` treats the most significant bit of the nonce as a special case, and to keep our signal analysis simpler we do not attempt to identify the signal snippet that corresponds to that bit or recover its value. Even with this additional unknown bit, in practice our signal analysis produces relatively few candidate nonces and thus leads to rapid recovery of the ECDSA private key.

3.4.5 Full Recovery of ECDSA Private Keys

Our recovery of the private ECDSA key considers each candidate nonce to determine if it produces a private ECDSA key that matches the known public key. Specifically, for each candidate nonce value k_c, we use the known values of z, r, s (z can be computed from the message that has been signed, and r and s are part of the ECDSA signature that has been computed) to compute a candidate private key d_c, using the equation

$$d_c = (sk_c - z)/r \; mod \; n$$

as described in Section 3.3.1. We then use this candidate private key d_c to calculate the corresponding public key $Q_c = d_c \times G$, and compare that public key Q_c to the publicly available actual public key Q_A of the signing entity (Alice). If Q_c is equal to Q_A, the signing entity's private key d_A is equal to the candidate private key d_c, and the analysis completes successfully. If q_c differs from Q_A, the next candidate nonce value is taken as k_c, the corresponding d_c and Q_c are computed, and the analysis continues until it either discovers the ECDSA private key d_A or runs out of nonce candidates.

In our experimental evaluation we have tested our signal analysis on 100 ECDSA signing operations, each with a different private key, and we have applied our signal analysis and full private-key recovery to each of these 100 ECDSA signing operations. We found that our signal analysis produces up to 2048 nonce candidates for each signing operation (i.e. up to 11 bits in the 256-bit nonce were unknown). The full recovery of the ECDSA private key was successfully completed for each of the 100 signing operations, which implies that in each set of candidate nonces the correct nonce was among the candidates, which further implies that every swap-related signal snippet that was identified in our experiments has produced a correct reconstruction for the $cond$ value used in that instance of the conditional swap operation. Overall, no signature required more than 1.3 seconds of analysis run-time to recover the correct private key, using only a single core on a moderately powerful laptop system (a Macbook Pro with an 2.7 GHz Intel Core i5 processor).

3.5 Experimental Evaluation

In this section we describe our measurement setup and results or recovering private keys from `GnuPG` and `OpenSSL` during ECDSA signature computation on three different devices.

3.5.1 Experimental Setup

We run `GnuPG`'s l`libgcrypt` and `OpenSSL` applications on two Android cell phones, Alcatel Ideal [51] and ZTE ZFIVE LTE2 [83], and on an A13-OLinuXino IoT development board [52]. The Alcatel Ideal has a quad-core 1.1 GHz Qualcomm Snapdragon processor with Android version 6, while the ZTE ZFIVE has a quad-core 1.4 GHz Qualcomm Snapdragon processor with Android version 6.0.1. The A13-OLinuXino is a single-board computer with an ARM Cortex A8 processor [53] with Debian Linux. For both `libgcrypt` and `OpenSSL`, we use the latest released version at the time we were working on this project – `libgcrypt` version 1.8.4, and `OpenSSL` version 1.1.1a. Both `libgcrypt` and `OpenSSL` use a constant-time implementation of point-scalar multiplication, relying on conditional swap operations to avoid control flow that depends on the bis of the scalar. We use `Ed25519` in `libgcrypt` and `ecp256k1` curve in `OpenSSL`. Both are commonly used curves, and for both the maximum order n is a 256-bit value, which also means that the secret nonce k used during ECDSA signing is at most 256 bits in size. We note, however, that the choice of the curve has no significant impact on our attack – ECDSA with any curve uses the same point-scalar multiplication to multiply the base point G with the secret nonce k, and this point-scalar multiplication is implemented as a sequence of steps, with each of these steps examining a bit of k and using the constant-time conditional swap whose condition directly corresponds to a bit in k (in `libgcrypt` of to the difference between the current and the previous bit in k (in `OpenSSL`). It should also be noted that, in addition to using constant-time operations, side-channel defenses in `libgcrypt` and/or

Figure 3.7: Experimental setup for receiving EM emanations from the ZTE ZFIVE (left) and Alcatel Ideal (right). The mechanical arm holds the custom probe (flat circular beige object at the end of the silver-colored cable held by the arm) close (but without touching) the phone, and the Ettus B200-mini SDR (white box) digitizes the signal and sends it through a USB cable to a personal computer (not shown) for analysis.

`OpenSSL` include message blinding and point blinding, which prevents the attacker from choosing the message and/or the base point in a way that increases side-channel leakage of information. However, message and point blinding have no effect on how the value of the nonce k is used in point-scalar multiplication, so they also have no effect on our attack.

Our setup for collecting the electromagnetic emanations from these devices is shown in Figure 3.7. It consists of a small custom electric probe to receive the EM signals, an Ettus B200-mini [84] software defined radio (SDR) to digitize the EM signal in the desired frequency band, and a personal computer to to process the digitized signals. As show in Figure 3.7, the probe is placed is close physical proximity to the target device, but without touching the device an without opening its enclosure. A mechanical arm is used to hold the probe in the desired position during the experiments. The probe is connected to the compact SDR which digitizes the signal and sends it, through a USB cable, to a personal computer (not shown in Figure 3.7) where the signal analysis and ECDSA key recovery is implemented as a custom program in MATLAB. Note that MATLAB is used mainly for

convenience, and that signal analysis and key recovery would likely be significantly faster if it were implemented in C/C++ on the personal computer. Even the need for a personal computer could likely be eliminated, at the cost of a significant hardware design effort, by implementing the analysis and key recovery as a custom hardware design within the Field Programmable Gate Array (FPGA) that is available within the B-200mini SDR itself.

3.5.2 Attack Results

Our experimental results are based on repeating the attack 100 times for `libgcrypt` and 100 times for `OpenSSL` on each device. In each attack, we first randomly generate an ECDSA private key and a message. Then we initiate signal collection while this ECDSA private key is used to sign the message. The signal from this single ECDSA signing operation is then analyzed to recover the nonce k and the ECDSA private key d_A, and finally the recovered d_A is compared to the actual d_A to determine whether the attack was successful. We note that we directly used the cryptographic API in both `libgcrypt` and `OpenSSL` to initiate the signing operation, rather than use a real-world application (like Apache for `OpenSSL` or Enigmail for `Libgcrypt`) to initiate a signing operation. However, since our attack identifies the part of the signal that corresponds to the EC point-scalar multiplication, this direct use of the cryptographic API has no material impact on the success of the attack, and we use direct API calls primarily because that substantially reduces the time needed to collect, store, and process the signals.

Our first set of experimental results consists of attacking a single instance of `libgcrypt`'s ECDSA signing operation, repeating this attack 100 times on each of the three target devices (ZTE, Alcatel Ideal and OLinuXino IoT board). All 300 of these attack instances were successful in recovering the ECDSA private signing key. More detail on the success of our attack is shown in Figure 3.8 where, for each device, we show the clustering of the signal snippets that correspond to the two values of the condition in the conditional swap, and we show the histogram for the number of candidate values for the nonce k that

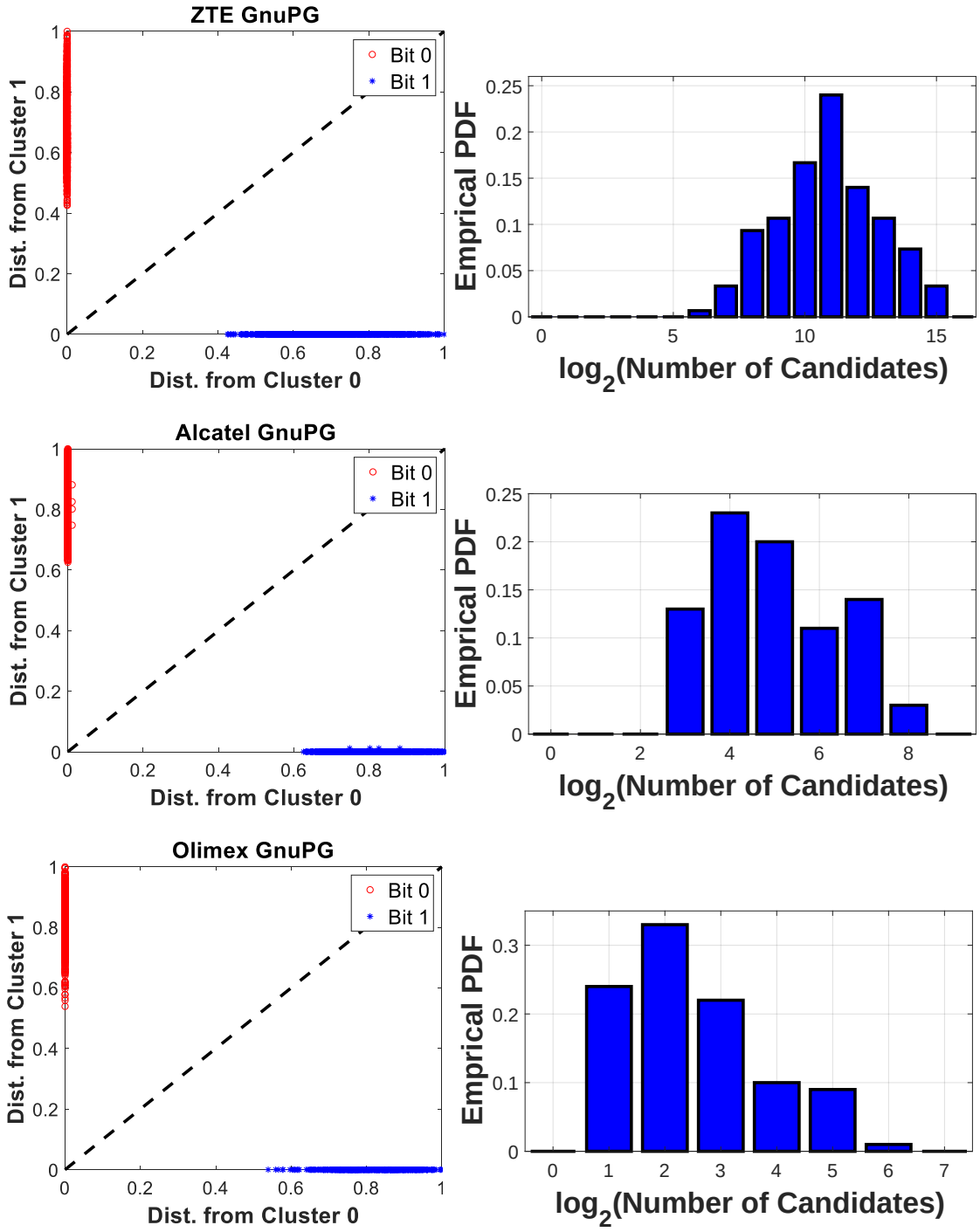

Figure 3.8: `Libgcrypt` detection of nonce bit based on the swap function clustering on bit 0 and bit 1 (left), `Libgcrypt` pdf distribution for the candidate nonce (right) for various devices.

were considered before the value of the ECDSA key was recovered. We can observe that, for all three devices, the signal snippets that correspond to the two possible values of the

swap condition are in well-separated clusters, so these clusters can be used to recover the value of the swap condition that was used during each signal snippet. We also observe that, for all three devices, the number of candidates that were considered is at most 2^{16} (for the ZTE). We also observe that for the OLinuXino board our attack typically needs to consider at most 2^6 candidate nonce values, that the number of candidates for the Alcatel Ideal is typically at most 2^8 or whereas for the two phones the typical number of candidates is larger. These differences are primarily caused by differences in how often there are occurrences of interrupts and other activity on these devices. Recall (Section 3.4.2) that we use timing to determine the position of the missing snippets, but that a missing snippet results in an unknown value of the swap condition at that position, and thus doubles the number of candidate nonce values that must be considered. We observe that on the two Android-based devices interrupts occur significantly more often than on the DEbian-based OLinuXino. Finally, on the ZTE phone the signal for Libgcrypt's point multiplication is unusually "choppy", resulting in sporadic failures to identify the swap snippets even in the absence of interrupts and other interference. Even on the ZTE, however, the number of candidates is so low that the worst-case analysis time was only 1.3 seconds when performing the analysis using only a single core in MATLAB on a moderately powerful laptop computer (a Macbook Pro with a 2.7GHz Intel Core i5 processor).

Our second set of experiments is for `OpenSSL`, where we also repeat the single-signing-operation attack 100 times for of the three devices. Figure 3.9 shows the clustering of signal snippets, and the number of candidate values for the nonce, for this set of experiments. We observe that, compared to `libgcrypt`, `OpenSSL` produces even more distinct clusters of snippets according to their value of the swap condition, i.e. the swap condition can be recovered from each signal snippet even more reliably than in `libgcrypt`.

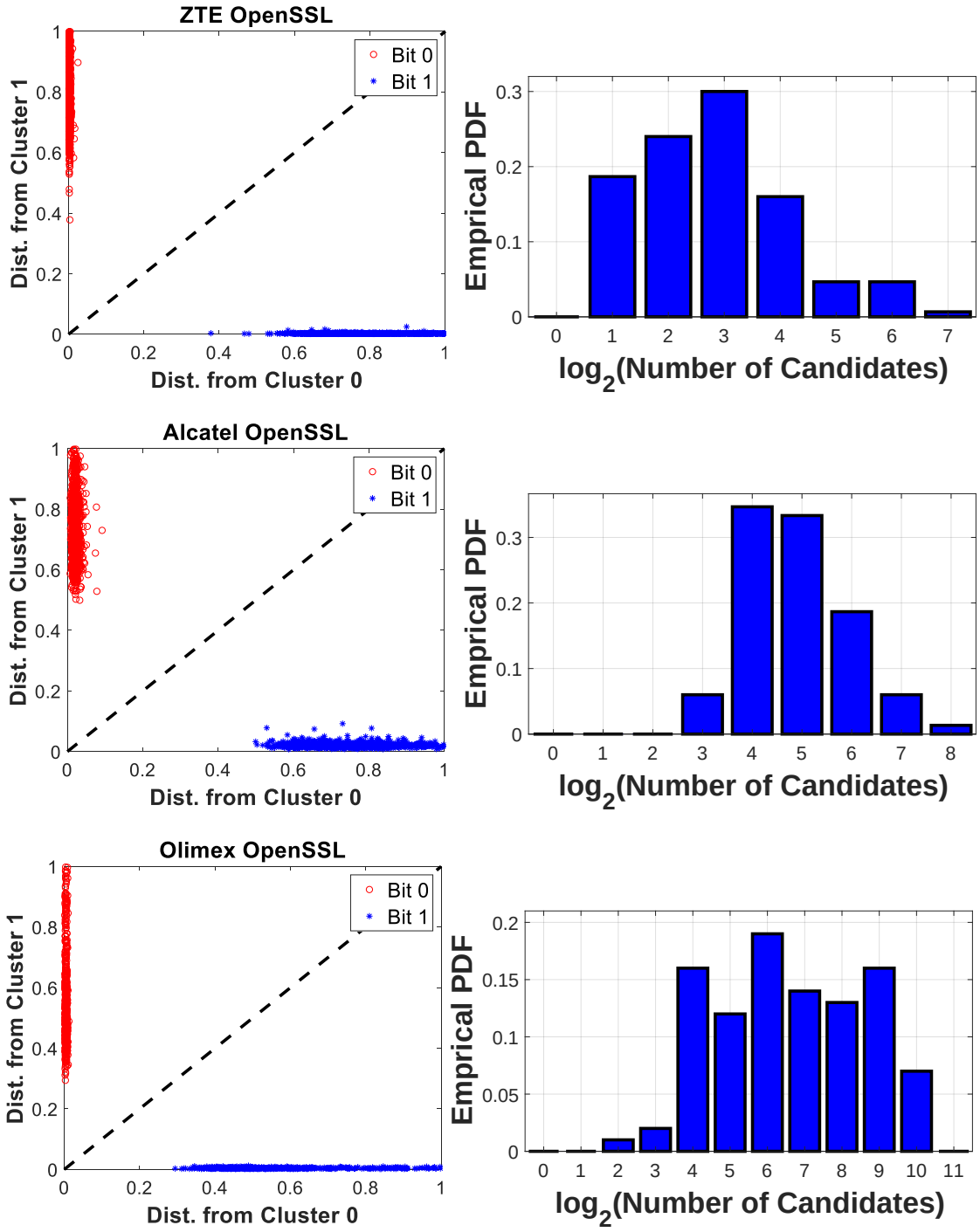

Figure 3.9: `OpenSSL` detection of nonce bit based on the `swap` function clustering on bit 0 and bit 1 (left), `OpenSSL` pdf distribution for the candidate nonce (right) for various devices.

3.6 Mitigation

Our mitigation is based on the key insight that, within the conditional swap operation, there are tens of machine words in the data structure that represents each of the two EC points, and to each of these machine words the conditional-swap function applies an exclusive-or whose other operand is either zero (when the swap condition is false) or has the value of the bit-wise change mask that changes that word's value to its new value. Although the difference in EM signals between an exclusive-or with a zero value and an exclusive-or with a semi-random value is very small, the conditional swap's signal performs tens of such operations, all with the same bias in their operands, and we believe that it is this systematic difference that enables the reliable clustering of conditional-swap signal snippets according to the value of the condition that was used.

```
1  EC_point_swap_cond(a,b,cond){
2    // When cond is 0, set mask to all-zeros
3    // When cond is 1, set mask to all-ones
4    mask=0-cond;
5    // Random value for mitigation
6    rand=random_word();
7    // For each machine word
8    for(i=0;i<nwords;i++){
9      delta = (a->w[i] ^ b->w[i]) & mask;
10     delta = delta ^ rand;
11     asm volatile(
12          // No actual assembler code
13         ";"
14         // But specify that rand is changed
15         : "+r" (rand)
16         : :);
17     a->w[i] = (a->w[i] ^ delta) ^ rand;
18     b->w[i] = (b->w[i] ^ delta) ^ rand;
19   }
20 }
```

Figure 3.10: `Libgcrypt` mitigation code where random bits are added into conditional `swap` function. The value of `delta` is masked to avoid systematic `cond`-dependent differences in exclusive-or operands.

To test this hypothesis and attempt to mitigate the vulnerability to the proposed attack,

we modify the source code of the conditional swap function shown in Figure 3.10. Specifically, we apply a random exclusive-or mask `rand` to the `delta` that is being applied to a pair of words via exclusive-or operations. For each word, the modified difference `delta` is applied (via an exclusive-or operation) first, and then the mask $rand$ is applied (also via an exclusive-or) to undo the effect of applying the mask to the original value of the difference. The net effect of this is that, when the condition has a zero value, the exclusive-or operations are all performed with the value of $rand$, while when the condition value is one, the exclusive-or operations are performed with the modified difference and then the value of $rand$. However, the modified difference and the $rand$ value both have unbiased values of their bits, i.e. 0-value and 1-value bits are equally likely, so different values of the condition no longer create the systematic difference in EM signals that the original implementation of conditional swap was producing. To leverage this change, we used gccs extended inline assembly syntax to tell the compiler that the effect of the rand value should not be removed.

We verify, as shown in Figure 3.11, that after this mitigation the clusters that correspond to the two values of the swap condition are no longer separable, and we further verify that our attack's recovery of the value of the swap condition, once the mitigation is implemented, has accuracy that is statistically equivalent to random guessing.

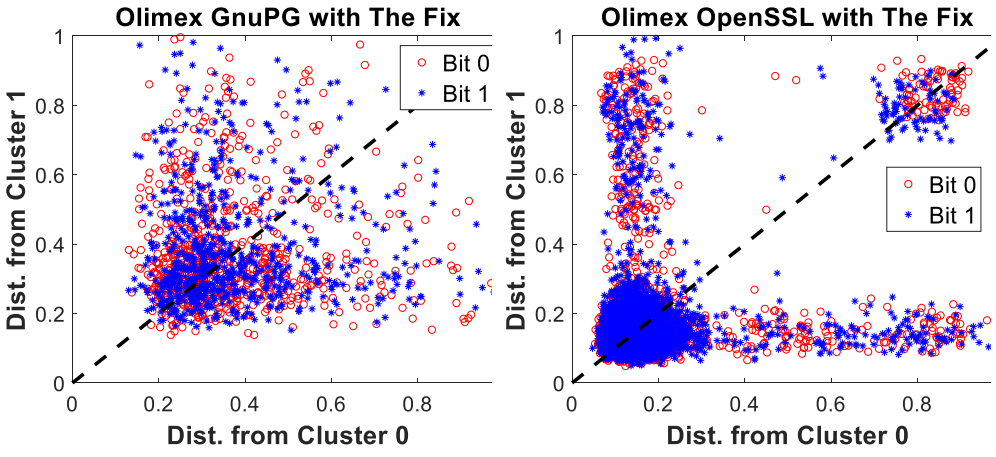

Figure 3.11: Detection of nonce bit based on the swap function clustering on bit 0 and bit 1 for `Libgcrypt` (left) and `OpenSSL` (right) after mitigation

Overall, we have experimentally confirmed that our proposed mitigation is successfully preventing the attack.

3.7 Related Work

Side-channel attacks [63, 64] have been demonstrated on numerous cryptographic implementations, and EM side channels have been exploited for attacking smart cards and other small devices [56, 57, 58]. Alenka *et. al* demonstrated the experimental setup [60] for emanated EM signal from modern processors, quantify [85, 86, 87] these leakages on different embedded devices, and detect malware [88, 89] by using spectral profiling [90]. The knowledge of these techniques are very important to implement our attack, in particular, our evaluation uses signals obtained by demodulating the signal from 40 MHz around the processor's clock frequency (around 1GHz).

Previous physical side channel attacks mentioned in the surveys [91, 92] targeted the ECC primitives implementations on small devices. Most of these attacks take advantage of naive implementations which contain key-dependent branches for `add` and `double`. Physical side channel attacks on complex devices are first demonstrated by Genkin *et. al* [30, 93] on RSA implementations. Different authors successfully demonstrated the EM based physical side channel attacks: on `ECDH` on PCs [72], on `ECDSA` on iPhone [73], on `ECDSA` on android smart phone [74]. All these attacks target the `OpenSSL's wNaf` based scalar by point multiplication. They used solver based techniques to retrieve full secret key.

`FLUSH+RELOAD` technique by Yarom *et. al* is applied to wide range of crypto system breaking (`OpenSSL` and `Libgcrypt`) by observing cache-side channel [75, 77, 78, 76]. It exploited the `wNaf` implementation of scalar by point multiplication part of these crypto benchmarks and revealed partial keys and broke full keys by using different solvers [94].

All these techniques are not valid any more (after 1.7.6 version released) when `Libgcrypt` enforce control-flow less and so-called constant time Montgomery ladder for `ECDH` and

control-flow less `add-double` with constant time implementation of `swap` function to exchange the values between two big numbers. `OpenSSL` initially enforced w-ary `wNaf` which was considered to be secured till their last stable release of 1.1.0h version. Although, `OpenSSL` very recently released their stable version 1.1.1 which enforce constant time Montgomery ladder in the place of `add-double` and use the same logic of `swap` function that was used by `Libgcrypt` to exchange the values between two big numbers so that correct result would be obtained without any control flow dependency. Unfortunately, Genkin *et. al* successfully break the `ECDH` on 1.7.6 version of `Libgcrypt` over the curve `Curve25519` [79]. They found that the field arithmetic operations for the `Curve25519` are not constant time implementation and they broke the key by using chosen cipher text.

As a countermeasure, the latest version of `Libgcrypt` (1.8.3) enforces point blindness as well as message blindness. All the signing operations execute on random digest and the generator `G` is masked with a random number to produce random base point such that all the point calculations happen on this random point. The attacker neither has ability to perform chosen cipher text attack nor to guess generator or base point to be operated on. Similarly, `OpenSSL` recently released the stable version where ECDSA signing default implementation introduces message and points blinding, Montgomery ladder and constant time `swap` for scalar by point multiplication. It is seemed that the crypto libraries are secure against all the previous techniques.

Recently, EM signal is used for control flow tracking [48] and it is applied for breaking secret key. Very recently, One&Done [95] breaks secret exponent d_p and d_q over constant time `OpenSSL's RSA` implementation with a single EM trace and retrieves full RSA key. The success of these type of attacks depends on the ability to capture demodulated signals around the clock and to identify a very small signal variation due to changes of data-flow over secret components. Our present work combines all the knowledge and techniques used by these two works. We emphasize on the fact that leakage not only depends on control-flow or different cache access for `add-double` or `Montgomery` ladder implementation,

but also depends on small data-flow over the secret bits, i.e., the `swap` function where processor behaves differently when it executes logical `AND` of 0/1 values with a big number. Hence, it emits EM signals with characteristic signatures which help us to reveal the bit the `swap` function is operating on.

We can classify our work as an example of Simple Power Analysis (SPA) [63], which is a classic power analysis technique and is a very simple but powerful attack comparing with Differential Power Analysis (DPA) [96]. Instead of power, we use EM as a side-channel information.

3.8 Conclusion

We present a new physical side-channel attack on ECDSA implementations of `GnuPG's Libgcrypt` and `OpenSSL` where so called control-flow independent constant time implementations are enforced. We have observed that there exist meaningful EM emissions during conditional `swap` of two big numbers (256-bit). Our attack is based on the electromagnetic (EM) emissions generated while the device performs cryptographic operations, such as digital signature and verification. We can split the proposed attack into two phases as training and detection. In the first phase, we train our system using different per-message keys, which are captured EM images of nonce-bit reading together with such `swap` for different keys followed by processing the EM signals in the time-domain. In the detection phase, with a single trace, we compare the images of conditional `swap` (bit-0 or bit-1) of test signals with trained signals. We found that our approach can detect $> 99\%$ secret nonce bits for constant time ECDSA implementations for the both crypto benchmarks. We also propose a countermeasure to hinder this vulnerability. With the proposed countermeasure implemented on the current `Libgcrypt` and `OpenSSL`, we could not separate the `swap` based on 0/1 bit computation. To evaluate the robustness and effectiveness of our attacks and corresponding countermeasure, we ported the latest stable version of `GnuPG's Libgcrypt` (1.8.4) and latest version of `OpenSSL` (1.1.1a) into two cell phones and one

embedded device, and executed applications that use the sign/verify libraries. We demonstrate that our approach works across different devices.

CHAPTER 4
RETHINKING DSA IMPLEMENTATIONS: A NEW ATTACK METHOD WITH A FEW EM TRACES

4.1 Abstract

While various optimization and counter-measures are enforced for public key crypto (PKC) implementations, there exist some so called ”glue-codes” which exhibit key dependency. Capturing emanated signals while executing these ”glue-codes” can lead to break PKC implementations. In that respect, we present a new physical side-channel attack on PKC implementations of OpenSSL. In particular, we consider DSA implementation as a use case, which utilizes constant-time fixed-window (m-ary) modular exponentiation. Our attack is based on the electromagnetic (EM) emissions generated while the device performs cryptographic operations, such as digital signature and verification. We have observed that there exist meaningful EM emissions during ”Window” computation for DSA sliding and fixed window implementations. We can split the proposed attack into two phases as training and detection. In the first phase, we train our system using different per-message keys, which are captured EM images of window computations for different keys followed by processing the EM signals in the time-domain. In the detection phase, with a single trace, we compare the images of window computations of test signals with trained signals. We found that our approach can detect 99% exponent bits for constant time DSA implementations. We also propose some counter-measures to hinder this vulnerability. With the proposed counter-measures implemented on the current OpenSSL, we could not detect more than 50% bits for fixed-window. We demonstrated different implementation aspects and their effects as countermeasures which embrace the importance of re-thinking before designing and implementing PKC, in general. To evaluate the robustness and effectiveness of our

attacks and corresponding counter-measures, we ported the latest version(at the time of our experiment) of OpenSSL (1.1.0g) to two cell phones and one embedded device, and executed applications that use the OpenSSL sign/verify libraries. We demonstrate that our approach works across different devices.

4.2 Introduction

Physical side-channel cryptanalysis is a very effective approach to break a secure crypto system. The attacks are based on signals generated from a processor while it carries out computations. These signals include electromagnetic emanations created by current flows within the computational and power delivery circuitry of a device [56, 57, 57, 60, 61], variations in power consumption during computation [62, 63, 64], acoustic [65, 30] and also temperature [66]. Prior physical side-channel attacks on PKC implementations rely on classifying signals corresponding to a large-integer square and multiply operations [29, 30, 31]. The focus on identifying this long-lasting subsequence was to identify overall sequence of samples corresponding to entire exponentiation. Also, these large integer square-multiply operations produce enough samples to classify them successfully even with relatively low sampling rate. The classification of these long-lasting square-multiplications would be difficult when the sequence of square-multiplications are not key dependent and when the attacker can not control the input message that would be exponentiated.

Keeping these in mind, open-source and commercial crypto libraries enforce many types of optimizations and counter-measures. For example, OpenSSL implements constant time Montgomery ladder to counter the square-multiply sequence identification. The ladder is implemented such a way that for both the square and multiply operations, it executes the same function with different parameters to ensure the constant execution time for either case. To counter cache timing attacks, OpenSSL executes constant time operations by accessing the cache table for pre-computed values. Intel developed a scatter-gather technique to enforce this constant time operation. These techniques, which prevent key

Table 4.1: Comparison between the proposed and other approaches, based on 1) which algorithm (sliding or fix window) it is applied on, 2) the applications (OpenSSH) and Library (like OpenSSL, GnuPG) used to break the PKC, 3) how many traces are required to break the key (normally it can take several traces to average the signals enough to eliminate the noise), 4) the target device that the attack is performed (the approach would be more acceptable if it attacks commonly used devices, such as cell phones, PCs, etc), 5) how many bits can be detected correctly (this is important for breaking full key without a brute-force attack).

Ref	**(1) Algorithm**	**(2) Target Application**	**(3) Number of Trace**	**(4) Target Devices**	**5 Accuracy** (%)
[31]	Sliding / Fix Window	OpenSSL	1433600	PC	100
[29]	Sliding / Fix Window	GnuPG	640	PC	100
[30]	Square and multiply	GnuPG	1000	PC	100
[97]	Fix Window	OpenSSL	16000	PC	60
This work	Sliding / Fix Window	OpenSSL	3 (Embedded Device) 10(Samsung) 50(Alcatel)	Cell Phones Embedded Device	100

exploitation presented in [29, 30, 31], are used when OpenSSL computes constant-time window exponentiation.

While there are various optimization considerations and counter-measures enforced for PKC implementation, there still exist some so called "glue-codes" which exhibit key dependency. Capturing and identifying emanated signals while executing these "glue-codes" make it possible to break PKC implementation. We successfully identify these codes and break constant time DSA implementation of OpenSSL on two cell phones and one embedded device. Comparing with other works, we exploit almost all exponent bits with a single EM trace and retrieve a full key with a few measurement traces (Table 4.1).

4.2.1 Our Contribution

Breaking Constant-time OpenSSL's DSA Implementation. We have seen that the Montgomery multiplication produces a unique and identifiable EM pattern which makes it

easy to isolate any bit of key computation at the edge of such multiplication. By obtaining enough training pattern, we are able to break 99% nonce constant time DSA signing operation with a single EM trace. We retrieve the full DSA private key out of the predicted nonce with maximum 22 EM traces, in worst case (section 4.4).

Counter-measure and Insight over PKC implementation. We also provide some counter-measure techniques and implemented these on top of the current OpenSSL. We demonstrated different implementation aspects and their effectiveness as countermeasure. This analysis embraces the importance of re-thinking before design and implementing PKC, in general (section 4.6).

Evaluation on Commercial Devices. We carried out an experimental evaluation by using OpenSSL native libraries and tested our attack on two cell phones: Android based Samsung smart phone and Android based Alcatel Lucent smart phone; and an ARM embedded device. The clock frequency for the tested devices are 1 GHz and, we observe that our approach works across all devices (section 4.5).

This chapter is organized as follows. Section 4.3 presents the background of DSA algorithm and it's implementation aspects of OpenSSL. Section 4.4 presents a new attack methodology. Section 4.5 provides experimental setup and the evaluation on different devices. Section 4.6 discusses some mitigations with experimental results. Section 4.7 presents the sliding-window implementation of DSA and section 4.8 concludes the chapter.

4.2.2 Targeted Software and Hardware

The targeted software is OpenSSL version 1.1.0g, the latest version at the time of our experiment. After mentioning a major bug on its DSA implementation by [70], and fixing this bug, OpenSSL's DSA sign uses constant-time fixed window exponentiation, constant time Montgomery multiplication and per-message random key. To counter the cache-timing attack, the present DSA implementation enforces scatter-gather technique to access random

cache line.

We demonstrated our attack on two cell phones: Android based Samsung smart phone and Android based Alcatel Lucent smart phone; and an ARM embedded device. The clock frequencies of all of the tested devices are 1 GHz.

4.2.3 Current Status of Mitigation

We notified the OpenSSL teams regarding this attack on the month of March and provided a fix on the month of May. OpenSSL made a patch with our fix and all of its current versions of OpenSSL. Therefore, its fork does not have this vulnerability as of now.

4.3 Background

In this section, we provide the general information about DSA implementation and corresponding notations used in this chapter.

Key generation The first phase is a choice of algorithm parameters which may be shared between different users of the system, while the second phase computes public and private keys for a single user.

Parameter generation: The first step is to choose an N-bit prime q and L-bit prime p such that $p-1$ is a multiple of q. The next step is to choose g such that the multiplicative order modulo p of g is q. This value can be obtained by setting $g = h^{(p-1)/q}\ mod\ p$ for some random h which is chosen from the interval $h\ in[1, p-1]$. It is possible to share the parameters (p, q, g) between different users of the system.

Per-user key: In the second phase of key generation, the implementation computes private and public keys for a single user. For that purpose, a secret key α is chosen by some random method, and a public key is computed as $y = g^{\alpha}\ mod\ p$. Here, x is specified to be in the interval $0 < x < q$.

Signing Assuming H is the hashing function and m is the message, signing operation is done as follows:

1. A per-message value k is generated by keeping $1 < k < q$ in mind.

2. The next step is to calculate $r = (g^k \ mod \ p) \ mod \ q$. However, the value of r must be calculated again for a different k if $r = 0$.

3. As the final step, the computation, $s = k^{-1}(H(m) + \alpha r) \ mod \ q$, must be performed. Similar to r, s must be calculated again with different k if $s = 0$.

Verifying During the verification, the first natural check is to control whether the conditions $0 < r < q$ or $0 < s < q$ are satisfied. If these conditions are not satisfied, the signature must be rejected. However, if these conditions hold, for the verification phase, the following mechanism must be checked: First, we need to compute $w = s^{-1} \ mod \ q$. Then, the intermediate parameters u_1 and u_2 must be calculated by $u_1 = H(m) \cdot w \ mod \ q$ and $u_2 = r \cdot w \ mod \ q$. The final operation is to obtain the candidate verification key v as $v = (g^{u_1} y^{u_2} \ mod \ p) \ mod \ q$. We conclude that the signature is valid if and only if $v = r$.

From the above operations, verifying does not leak meaningful information to obtain private key α. During key establishment, given the public key y, and parameters g and p are known, calculating the private key α requires solving a complex discrete logarithm problem. Moreover, previously introduced side-channel attacks may not be feasible here because the private-public key pair is generated once for a session for a given user. The signing operation is the soft target to obtain private key α. People normally targets modular exponentiation operation (at item (2) of Example 4.1) to obtain the random nonce k first and calculate r from their attacks as all others parameters like p, q and g are public. So, when r and k are known, the remaining part is just solving the well known mathematical equations to get the private key k from the item (3) of Example 4.1 as all other parameters including signing message s is known. OpenSSL enforces many optimizations and counter-measures while computing the signing operation, in particular, the exponential multiplication which is the most costly operation.

4.3.1 DSA in OpenSSL

Here, we will focus only on signing part which is the potential target to break private key. For that purpose, let us examine DSA a bit further. During signing operation, the random nonce k is generated before computing r, where $1 < k < q$. The FIPS (Federal Information Processing Standards) 186-3 specifies the length pairs L and N as (1,024, 160), (2,048, 224), (2,048, 256), and (3,072, 256). So, the max length of q, hence for the k, would be 256-bits. To prevent the timing attack mentioned by [31], OpenSSL adds k with q or $2q$ to make k constant bit-length. The aim is to avoid time information leakage due to length of k. Then, it computes the complex exponentiation operation, i.e, $g^k \ mod \ p$. The old and naive

```
1 for (i = 1; i < bits; i++) {
2     if (!BN_sqr(rr, rr, rr, ctx))
3         goto err;
4     if (BN_is_bit_set(k, i)) {
5         if (!BN_mul(rr, rr, v, ctx))
6             goto err;
7     }
8 }
```

Figure 4.1: OpenSSL exponential multiplication

implementation of so called square-multiply reads each bit of k and does a square operation and perform only multiplication operation if the corresponding bit is 1 (as shown in Figure 4.1). The function `BN_is_bit_set` is used to check whether the i^{th} bit of k is set or not. Based on i^{th} bit, the function returns the value 0 or 1. In the naive implementation above, occurrence of squaring tells the attacker that the next bit of the exponent is being used. Moreover, an occurrence of multiplication indicates the value of the interested bit is 1. Therefore, an attack, which correctly recovers the square-multiply sequence, can trivially obtain all bits of the secret exponent. To overcome key dependent branching and prevent isolation between the two functions `BN_sqr()` and `BN_mul()`, OpenSSL substitutes these two functions with a Montgomery ladder by implementing `BN_mod_exp_mont()` function. In this function, all operations are performed in Montgomery form, and the results are converted back to the standard representation. With this technique, it reduces some per-

formance overhead and prevents the isolation between square and multiplication to some extent.

Even with Montgomery multiplication, the vast majority of execution time for large-number exponentiation is spent on large-number multiplications. So, optimization techniques to improve performance focus on reducing the number of these multiplications. Likewise, most of the side channel measurements (e.g. signal samples) are collected during large-number exponentiation which corresponds to large number multiplication activity. Hence, existing side channel cryptanalysis approaches tend to target multiplication activity. One classical example for this type of attack is FLUSH+RELOAD [68], which exploits instruction cache behavior.

To improve performance, most modern implementations use window-based exponentiation. For this approach, squaring is needed for each bit of the exponent, however, complex multiplication operation is needed only once for each multi-bit group. Every time OpenSSL computes a DSA signature, the exponentiation method `BN_mod_exp_mont` in crypto/bn/bn_exp.c is called. Here, based on the `BN_FLG_CONSTTIME` flag value, either a fixed-window or a sliding-window operation is performed. Then, for finite field operations, `BN_mod_exp_mont` calls `BN_mod_mul_montgomery` in crypto/bn/bn_mont.c. After that, the multiply wrapper `bn_mul_mont` is called, where assembly code is executed to perform low level operations using BIGNUMs for square and multiplication by default for x64 targets. Since squares can be computed more efficiently than multiplication, OpenSSL's multiply wrapper checks if the two pointer operands are the same and, calls the assembly squaring code (bn_sqr8x_mont) if the wrapper returns true. Otherwise, it calls the assembly multiply code (bn_mul4x_mont).

Figure 4.2 shows the sliding-window implementation of OpenSSL version 1.0.g In this algorithm, a squaring (lines 5 and 29) is performed for each bit while the multiplication operation (line 32) is performed only at the (1-valued) LSB (Least Significant Bit) of each window. After the timing attack described in [31], OpenSSL implemented these multi-

```
1  for (;;) {
2      //Chek if exponent bit == 0;
3      if (BN_is_bit_set(p, wstart) == 0) {
4          if (!start) { //Square
5              BN_mod_mul_montgomery(r, r, r, mont, ctx);
6          if (wstart == 0)
7              break;
8          wstart--;
9          continue;
10     }
11     j = wstart;
12     wvalue = 1;
13     wend = 0;
14     //Scan all bits in window length
15     for (i = 1; i < window; i++) {
16         if (wstart - i < 0)
17             break;
18         if (BN_is_bit_set(p, wstart - i)) {
19             // Increment window if it is set
20             wvalue <<= (i - wend);
21             wvalue |= 1;
22             wend = i;
23         }
24     }
25     j = wend + 1;
26     if (!start){
27         //Loop iteration windowsize
28         for (i = 0; i < j; i++) { //Square
29             BN_mod_mul_montgomery(r, r, r, mont, ctx);
30     }
31     //multiplication by table[windowValue]
32     BN_mod_mul_montgomery(r, r, val[wvalue >> 1], mont, ctx
           );
33     wstart -= wend + 1;
34     wvalue = 0;
35     start = 0;
36     if (wstart < 0)
37         break;
38 }
```

Figure 4.2: OpenSSL sliding window implemented in `BN_mod_exp_mont()` function at *crypto/bn/bn_exp.c* file. We provide a segment of code where the exponentiation executes

plication routines in such a way that the execution time is barely dependent on the input parameter passed to the multiplication function `BN_mod_mul_montgomery()`. But, at

the time of multiplication, a precomputed table is read to pass the multiplicand. Exploiting cache behavior, the sequence of a partial key is revealed by [67, 68], and combining with other techniques as given in [42] helps to reveal full key from this partial key.

```
1   // Scan the exponent one window at a time starting from
        the most
2   // significant bits.
3  while (bits >= 0) {
4      wvalue = 0;      /* The 'value' of the window */
5
6      /* Scan the window, squaring the result as we go */
7      for (i = 0; i < window; i++, bits--) {
8          if (!BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont,
               ctx))
9              goto err;
10         wvalue = (wvalue << 1) + BN_is_bit_set(p, bits);
11     }
12     // Fetch the appropriate pre-computed value from the
           pre-buf
13     if (!MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top, powerbuf,
           wvalue,
14                                                 window))
15         goto err;
16     /* Multiply the result into the intermediate result */
17     if (!BN_mod_mul_montgomery(&tmp, &tmp, &am, mont, ctx))
18         goto err;
19 }
```

Figure 4.3: penSSL constant time window implemented in `BN_mod_exp_mont_consttime()` function at crypto/bn/bn_exp.c file. We provide a segment of code where the constant-time exponentiation executes.

Concerns about the exponent-dependent square multiply sequences have led to adoption of fixed window exponentiation in OpenSSL, which combines the performance advantages of window-based implementation with an exponent-independent square-multiply sequence. OpenSSL enforces constant time cache access for Montgomery multiplication during fixed-window computations. A segment of the code is presented in Figure 4.3, where the window is computed at line 10. This window value is passed to `MOD_EXP_CTIME_COPY_FROM_PREBUF`, where the access cache is randomized. So, Percival's attack [67] does not effective here. Although, Yarom *et al.* [97] break partial exponent bits in presence of ran-

domized cache enforcement, they have done thousands of measurements. Recently, Yarom *et al.* [70] identify a potential bug of OpenSSL's `BN_FLG_CONSTTIME` flag setting for DSA signing operation. With this bug, the original control flow goes to sliding window exponentiation instead of executing constant time fixed-window exponentiation. They effectively break DSA private key using thousands of measurements. OpenSSL already adds a patch to fix this bug and current DSA signing operation correctly executes the constant time fixed-window code.

In summary, OpenSSL enforces optimization in many ways to reduce computation time for big exponentiation. They enforce different security techniques to counter against many attack types. They mainly focus on big number multiplication computation, different cache read, etc. To bring these security and optimization techniques, they introduce some small logics. We call these as "glue-codes", which are executed relatively very brief period of computation. By identifying this small period of computation, we break secret exponent bits. For example, we identify signal when computing window value (line 10 Example 4.3) for fix-window. Similarly, observing the execution of small computations, we identify when the window starts, the window end, the window value, etc. in case of sliding window. In the next section, we discuss how we break the key by utilizing the information leakage through these tiny operations.

4.4 A New Attack

In this section, we present a new technique to get secret exponent used in DSA through the signing operation. After fixing the bug reported by [70], OpenSSL ensures that the signing operation always executes constant time exponentiation path. That is why our focus is on the constant time signing operation which will be explained in detail in the following (sub)sections.

4.4.1 EM Data Acquisition

Unintentional EM emanations occur at various frequencies, but of particular importance is the frequency band centered around the clock frequency of the device's processor and memory. The frequency band contains signals that are primarily a function of the instruction sequence executed by the CPU. Each processor cycle, the CPU draws a current which is a direct result of the instruction(s) being executed. Much of this instruction-dependent current is drawn by the CPU clock circuitry and by circuitry which does new computations (i.e. switches on and off) every CPU clock cycle. This creates a strong current at the CPU clock frequency which acts as a carrier modulated by the clock-to-cycle variations in program activity (i.e. executed instructions). At the CPU and memory clock frequencies (and their harmonics), the generated EM emanations can propagate far enough to be observed with a high signal-to-noise ratio. Observing the leaked signal as described in this section, the emanating device has much in common with a communication system. The reason behind is that the device behaves like a transmitter (inefficiently and unintentionally) which transmits a message signal carrying information about program activity using a carrier signal (i.e. the clock signal). We can then receive and demodulate this signal using some techniques which are exactly same with the ones used in wireless communications. All EM signals, both in the training phase and in the monitoring phase, are demodulated with respect to the processor clock frequency, low-pass filtered, and sampled before being sent for signal processing. We process the signals in MATLAB.

4.4.2 Signal Processing

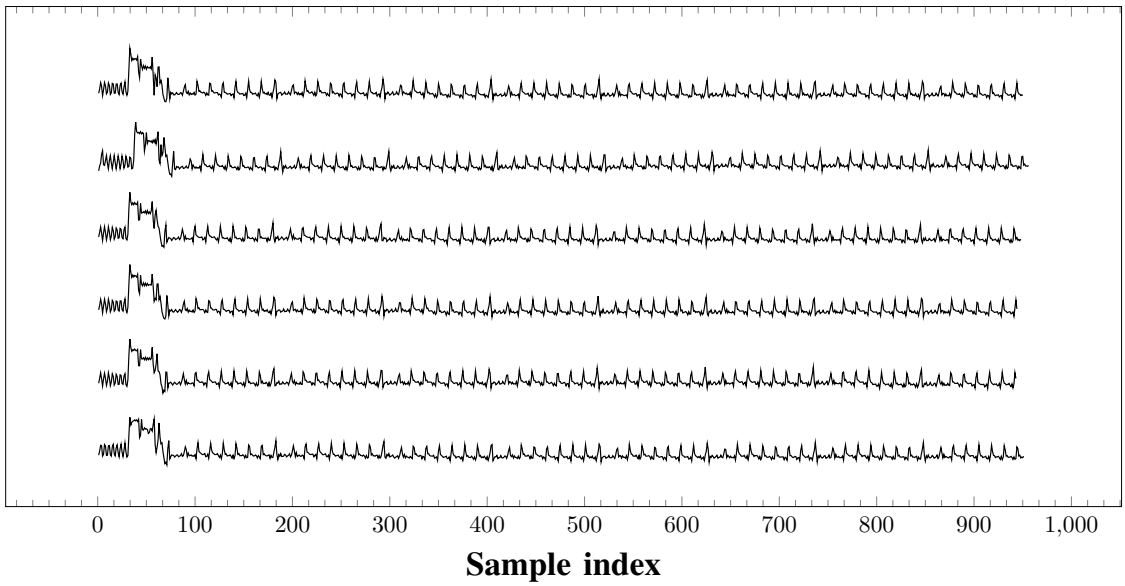

Figure 4.4: The EM signatures of $BN\_mod\_mul\_montgomery$ function, which take almost same amount of time irrespective of input.

The first part of the signal processing is to identify the sign/verify area where the window exponentiation takes place. The major operations of this part is Montgomery multiplication in repeated ways. Detecting where the exponentiation happens can lead to reveal the bits of nonce key k, which is the only unknown to unveil the private key. In that respect, capturing EM signals during signing operation, and attaining the locations, where the exponent computation happens, is crucial. We observe that the EM signal for each exponentiation follows a clear pattern which looks similar and takes similar amount of time as expected. Keep in mind that, OpenSSL uses the same multiplication function ($BN\_mod\_mul\_montgomery()$) for square and multiply with different parameter passing. This property of OpenSSL demonstrates a unique and visually discernible EM signal of each function call. We can easily detect this function through visible inspection which is shown in Figure 4.4. In this figure, we plot a portion of the EM signal captured during message signing and some of the exponentiation signals. It is clear that the number of

samples for each operation is almost same.

The multiplication does not help to predict the nonce directly, but it helps to identify the EM signal locations where either control-flow branching or window value calculation take place. So, we first need to isolate the signals for Montgomery multiplication $BN\_mod\_mul\_montgomery()$.

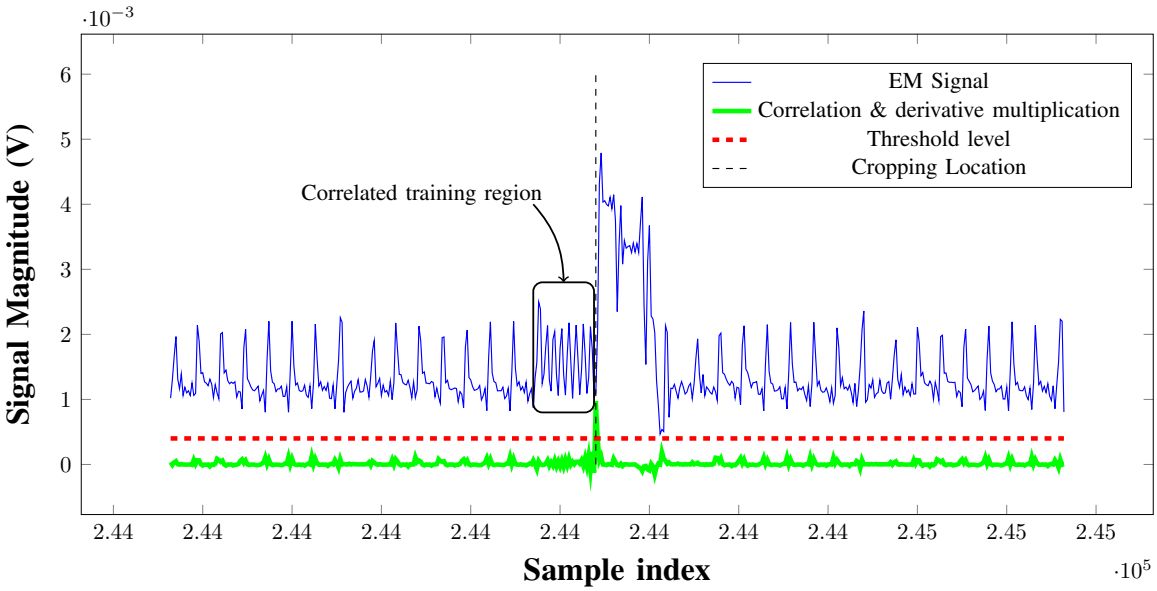

Figure 4.5: The edge of Montgomery Multiplication constitutes a unique pattern

The first feature of the signal of interest is that it encounters with a sudden increase at the edge which can be exploited to gather exponentiation locations as seen in Figure 4.5. However, utilizing only this feature of the signal does not provide enough information to detect the edges accurately. Another feature we observed is that there exists a unique signature signal before the edges where the signal exhibits a sudden increase. This signature signal is shown in the dashed black box in Figure 4.5. As the amplitude of EM signal for this pattern could vary for different multiplication, we create a template for this pattern where the pattern for multiplications contains 30 samples. Combining both of these features helps us to find the edges of the signal of interest successfully. For that purpose, we first calculate the derivative values of the captured EM signal at each sample point

and determine the candidate edge points by comparing their derivative values with a given threshold. Later, we check whether these candidate edge points are locally maximum. We eliminate the ones which have comparably small value in a given locale. As the second step, we check the correlation of the candidate edge signals with the signature signal given inside the dashed black box in Figure 4.5. We calculate the correlation of this template with the measured signal at the candidate edge points to identify all edges in a signing operation. After applying a threshold to the correlation values of the candidates, the remaining points are assigned as the edges of the signals of interest.

It is important to identify all the multiplication edges without any false positive or any false negative, as a single false positive or false negative will result random prediction for the key. We find that more than 99% time we can detect the multiplication edge correctly. We evaluate the correct edge detection for each signing by a *magic* number. Remember that, OpenSSL add q or $2q$ with randomly generated nonce k and makes the k with exactly q-bit length (256-bit in our case), so that there will not be any timing information for different k for different signing operations. As the window size (defined by OpenSSL) is 4 for 256-bit exponent k, that means there will be consecutive four squares and there will be a single multiplication after each 4^{th} square operation. That means we should have exactly 320 $(256 + 256/4)$ multiplication for the calculation of exponentiation part for a single signing operation. So, we evaluate the edge detection correctness with this number and we completely eliminate those signatures where we detect some more or less (which is the case of 1%) edges.

4.4.3 Recovering Nonce k

When we compute the multiplication edge correctly, this signifies that we locate the signal that corresponding to window value calculation and this window value is nothing but the exponent bits. As shown in Example 4.6, the sequence of the operations are: large-number multiplication used for squaring (line 10), then compute window value (line

```
1  int BN_is_bit_set(const BIGNUM *a, int n)
2  {
3      int i, j;
4      bn_check_top(a);
5      if (n < 0)
6          return 0;
7      i = n / BN_BITS2;
8      j = n % BN_BITS2;
9      if (a->top <= i)
10         return 0;
11     return (int)(((a->d[i]) >> j) & ((BN_ULONG)1));
12 }
```

Figure 4.6: OpenSSL BN_is_bit_set() function which check if a given bit is 0 or 1

12) followed by fetching pre-computed value from a table, and finally updating the result through large-number multiplication (Line 21). So, we essentially have three transitions here: square-windowComputation-square, square-windowComputation-tableFetch and multiplication-square. This transition happens in a unique pattern. For example, for 256-bit DSA signing operation, window value is 4 (OpenSSL defines default window values for different exponent bits for performance optimization). The sequence of transitions are: square-windowComputation-square, square-windowComputation-square, square-windowComputation-square, square-windowComputation-tableFetch, multiplication-square. The sensitive information, i.e window value of nonce k is executed when the transition has windowComputation. So, we are only interested in the transitions of square-windowComputation-square and square-windowComputation-tableFetch. We can isolate these transitions easily as every 4^{th} transition is a table look-up and every 5^{th} transition is a multiplication. After isolating the transitions, we are only interested at the edge where the windowComputation happens. The window is computed at the edge of multiplication. As we have seen that the isolation of this multiplication is quite easy, we can claim that small snippet corresponding to only windowComputation is trivial.

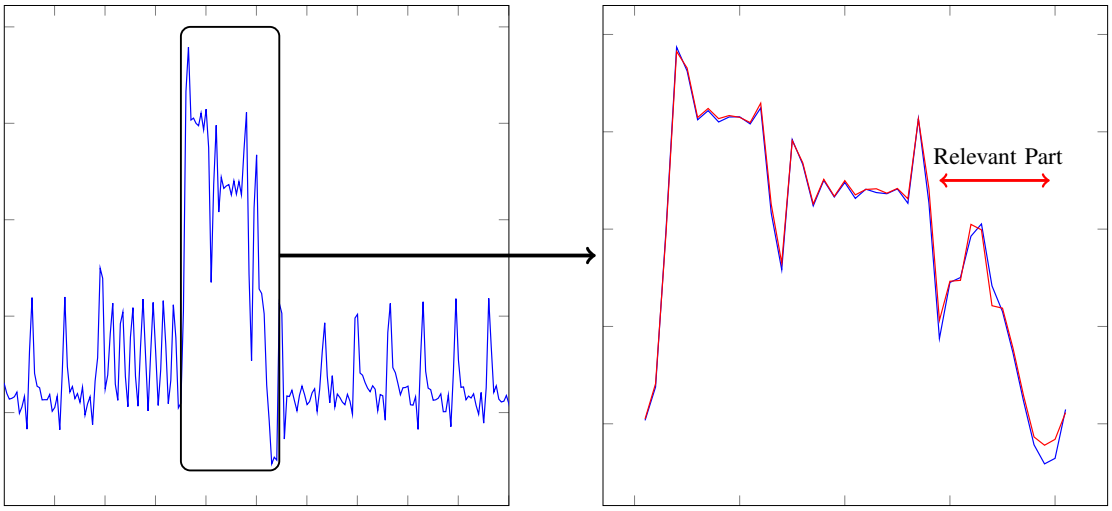

Figure 4.7: The centroid for two different snippets when the window is computed for 0 and when the window is computed for 1

We first capture this snippet which corresponding to window computation. The window is always shifted left and added with 0/1 based on the return value of $BN\_is\_bit\_set$ function. This function check a bit of a BIGNUM which represents by array of machine words. To read the n^{th} bit value, first it locate the memory word location (line 7) and then it identify the bit position (line 8) in that memory word (Example 4.6). To calculate the value of that bit, it is right shifted by its potion so that the bit becomes the least significant position of that machine word (line 11), followed by masking with 1. We found that this function together with addition of the return value have different signatures while it computes the bits 0 and 1. To account for other value-dependent segments in the signal, in each such set of snippets we cluster similar signals together and use the centroid of each cluster as the reference signal. We use the K-Means clustering algorithm and the distance metric used for clustering is Euclidean distance (sum of squared differences among same-position samples in the two snippets). Due to noise propagation and data dependency from different parts, we observe that snippet for 0 computation itself has different signatures, which is also same

for the computation of 1 as well. So, we make 10 sub-cluster for 0 and 10 sub-cluster for 1 bit computations. We calculate the centroid of each sub-cluster. Increasing the number of sub-clusters can cause overfitting or overlapping of two centroids corresponding to bits 0 and 1. Therefore, the number of sub-clusters is chosen carefully to differentiate between the centroid of 0 computation and the centroid of 1 computation and to avoid the given problems. This differentiation can be identified successfully in a high bandwidth signal given SNR is high enough (Fig 4.7).

4.5 Evaluation

4.5.1 Experimental Setup

We run the OpenSSL application on Android cell phones like Alcatel [51] and Samsung [50], and an IoT device (A13-OLinuXino board [52]). The Alcatel has quad-core 1.1 GHz Qualcomm Snapdragon processor with Android OS (version 6) and the Samsung has single-core 800 MHz Qualcomm MSM7625A Chipset with Android OS (version 5). The A13- OLinuXino board is a single-board computer that has an in order, 2-issue Cortex A8 ARM processor [53] and runs Debian Linux operating system. It is commonly used as a platform for prototyping IoT systems.

Our experimental setup receives EM signals by a lab made small electric probe. We place the probe very close to the monitored system (see Fig. 3.7). The signals collected by the probe are recorded with an Infiniium S-Series Oscilloscope (Keysight DSOS804A [54]). Our decision to use an oscilloscope was mainly driven by its existing features, such as built-in support for automating measurements, saving and analyzing measured results, visualizing the signals when debugging code, etc. The detection algorithm was implemented in MATLAB and run on a personal computer.

4.5.2 Results

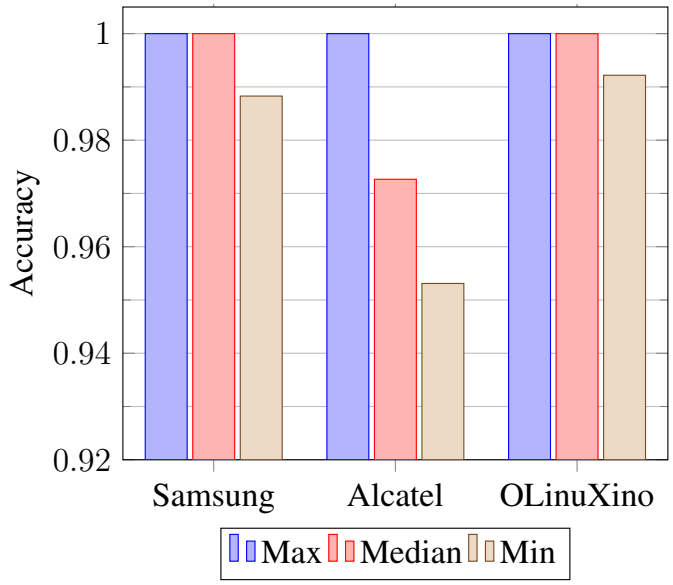

Figure 4.8: DSA nonce k prediction accuracy for Fix-window

To evaluate the effectiveness of our approach, we selected the latest version of OpenSSL (1.1.0g, at the time of our experiment), all its versions and forks (e.g. LibreSSL and BoringSSL). We choose (L, N) pair as (3072, 256), so the max length of q, hence for the k, would be 256-bit. We create an application that uses OpenSSL crypto library through standard API provided by OpenSSL and all parameters and public-private pairs are generated through command line utilities (like: # *openssl dsapair* , # *openssl gendsa*, etc.).

We first check how efficiently we can detect the nonce k. The nonce is randomly generated for each message signing and it is unknown but *known to be unknown* in our evaluation process. We use 50 signing instances for a given private key for training purposes for each devices. We use 50 randomly generated nonce for training. Each signing generates 256 snippet and total 256*50 snippets are used for training. Now we perform the actual testing where we detect unknown but *known to be unknown* nonce. We define our accuracy as the fraction of exponent we correctly recover. For a given DSA private key we sign 50 different messages and collect 50 different EM traces for these 50 signing operations. For each

signing operation, we found most of the bits correctly detected and we miss maximum 2-bit (99.2% worst case accuracy) for OLinuXino and maximum 3-bit for Samsung as shown in Figure 4.8. We found the worst case accuracy (missed maximum 12 bits) for Alcatel cell phone is around 95%. This is mostly because of activities on the other three cores interferes with the activity on the core doing DSA signing. But, in the best case situation, we found at least one correct nonce. This situation happens when the device does not have any interrupt at the time of signing (this happens in a fraction of second) and when snippet identification can not be ruined by other's core activity. We found that we get at least one correct nonce across all devices, and this is really important to retrieve private key.

4.5.3 DSA Key Recovery

There is a direct relation between the secret key and the calculated nonce which is given as

$$\alpha = (sk - z)/r \; mod \; q$$

where all are known except k which is predicted from the signal snippet. We calculate the private key α by using the nonce k predicted from our signal processing. Signing multiple messages under the same key is common when the key is fixed by a public key infrastructure. The correct key is obtained as follow:

- Estimate different k for each signing with a given α,
- Obtain the candidate α by inserting the estimated nonce k,
- Repeat the process until at least two candidate α's are exactly same with each other.

The idea for the procedure above is that as the set of all possible values of the private key are too large, obtaining two equal values for the private key given the incorrect estimation of two different nonce k is highly unlikely.

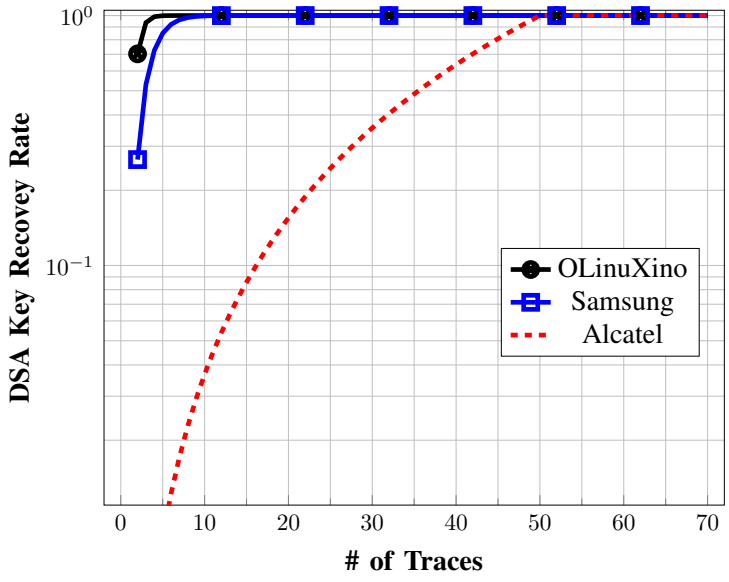

Figure 4.9: DSA private key prediction accuracy for Fix-window.

We implement the above relation in python by using python crypto library and calculate α by using known values s, z, r and q and predicted value k. To see how robust our technique is, we test 50 DSA private keys. We break 50 keys with 10 signing operations for Samsung, 3 signing operations for OLinuXino, and 50 signing operations for Alcatel on average. A single key is broken in 2 seconds (average) on mac-book pro with 2.7 GHz Intel Core i5.

4.6 Counter-measure

We focus our mitigation efforts on the fixed-window implementation, which is the implementation of choice in the current version of OpenSSL, and already mitigates the problem of exponent-dependent square-multiplication sequences, and timing variation. Some potential mitigations thus include circuit-level approaches that reduce the effect of the differences in computation, additional shielding that attenuates the emanated signals to reduce their SNR ratio outside the device, deliberate creation of RF noise and/or interference, etc. We do not focus on these mitigation techniques because they increase overall cost, weight, and/or power consumption of the devices. Moreover, these techniques are difficult to apply

to devices that are already in use, and may not provide protection against the side channel attacks due to emanated EM signals but through a different physical side channel (e.g. power).

An alternate approach to mitigate the attack is to minimise the attackers ability to precisely locate the brief snippets of signal, which correspond to examining the bits of the exponent during an exponentiation operation, and constructing the value of the window. For that, our main focus is on different implementation techniques for the OpenSSL library to circumvent the attacks based on emanated EM signals. In the following subsections, we thoroughly discuss the possible techniques and their results obtained from experimental demonstrations.

4.6.1 Window compute randomization

```
1  int BN_is_bit_set_randomized(const BIGNUM *a,
2                               int n, int wsize, int rnd) {
3    int i, j;
4    // Set all bits to 1 except the least significant wsize
       bits
5    BN_ULONG rmask = ~((1<<wsize)-1);
6    bn_check_top(a);
7    if (n < 0)
8      return 0;
9    i = n / BN_BITS2;
10   j = n % BN_BITS2;
11   if (a->top <= i)
12     return 0;
13   /* Randomize all bits except the LSB (0-th position),
14     then sero out bit in positions 1 through wsize - 1
15   */
16   return (int)( ((((a->d[i]) >> j))
17            ^ (rnd & (~1))) &((BN_ULONG) (rmask+1)) );
18 }
```

Figure 4.10: penSSL modified BN_is_bit_set() function which check if a given bit is 0 or 1 in presence of random mask.

In this approach, we try to mitigate the attackers ability to distinguish between the signals whose computation has the same control flow but uses different values for the ex-

```
1  /*
2  We compute window value as 32-bit value and then cut
3  the window length value form that
4  */
5  //Generate 32 words of random numbers
6  if(!BN_rand(rnd, 1024, BN_RAND_TOP_ANY,
7                BN_RAND_BOTTOM_ANY)) {
8    goto err;;
9  }
10 while (bits >= 0) {
11   wvalue = rnd->d[bits%32] ; /* a 'random' value of the
         window */
12   for (i = 0; i < window; i++, bits--) {
13     if (!BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont, ctx)
         )
14       goto err;
15     // Calculating window in presence of random value
16     wvalue = (wvalue << 1) +
17       (BN_is_bit_set_randomized(p, bits, window, rnd->d[
           bits%32] ));
18   }
19 /*
20 Un-mask the window val first and then
21 fetch the appropriate pre-computed value from the pre-buf
22 */
23 if (!MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top, powerbuf,
24                       (wvalue & ((1<<window) -1) ), window))
25   goto err;
26 }
```

Figure 4.11: OpenSSL modified constant time window computation in BN_mod_exp_mont_consttime() function, where window is computed in presence of random mask and un-mask the window value before using it.

ponents bit. In this regard, the attack benefits significantly from the limited space of possibilities for the value returned by BN_is_bit_set there are only two possibilities: 0 or 1. Based on this observation, we create a mitigation approach that puts a random value into the upper bits (those not used for the actual window) of the value returned by BN_is_bit_set itself (line 17 of Figure 4.10), which in turn creates a much larger space of possibilities for the operands of instructions that use BN_is_bit_sets return value and the value of wvalue itself, and yet allowing the correct value of wvalue to be recovered by masking out those

higher bits (line 24 of Figure 4.11). Since these random bits outnumber the exponents bits, in each of the values where exponent bits are present, the variation in signals caused by differences in these random bits is significantly stronger than the variation due to the values of exponents bits.

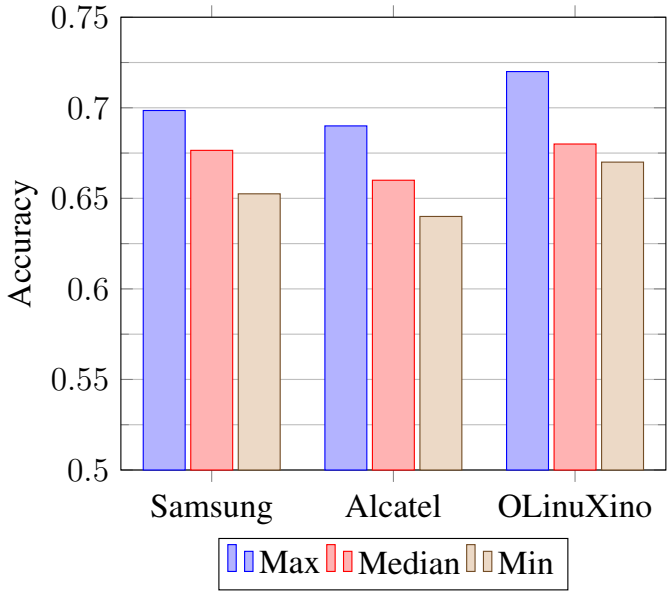

Figure 4.12: Bit prediction accuracy of nonce k when we add randomness in bit reading and window value computation.

This mitigation adds only a few instructions for each bit of the exponent, so it increases the overall executing time of the exponentiation by only 0.4% while, the recovery rate for the exponents bits decreases to between 60% and 70% as shown in Figure 4.12. Although we reduce the accuracy from 100% to 70%, it is still high (note that a 50% rate or recovery is equivalent to randomly choosing the recovered value of each bit). The possible reason for this high accuracy in the presence of randomization is that each iteration window is computed for a single bit, and there are only two possibilities (0 or 1) for the unknown variable. Our k-mean algorithm is strong enough to choose the correct cluster to identify the value of executed bit and, there are other parts of the code which is unchanged which contributes to the overall signal signature. Moreover, unbiased predictions of a variable with two possibilities always have 50% accuracy with random guess, therefore, with k-mean technique

and with high data accusation device, improving accuracy 20% is not a surprise. So, we need to increase the possible prediction window rather than two possibilities (0 and 1), for that we should not compute single key bit while computing w-bit window of key.

4.6.2 Never compute single bit window

The idea is that if we manage to compute a chunk of key, for example, w-bits (in this case, w = 4) then the range of prediction window becomes 2^w. OpenSSL already implements a function to compute word length window of a key as defined in Example 4.13.

```
1  //#if defined(SPARC_T4_MONT)
2  static BN_ULONG bn_get_bits(const BIGNUM *a, int bitpos)
3  {
4    BN_ULONG ret = 0;
5    int wordpos;
6    wordpos = bitpos / BN_BITS2;
7    bitpos %= BN_BITS2;
8    if (wordpos >= 0 && wordpos < a->top) {
9      ret = a->d[wordpos] & BN_MASK2;
10     if (bitpos) {
11       ret >>= bitpos;
12       if (++wordpos < a->top)
13         ret |= a->d[wordpos] << (BN_BITS2 - bitpos);
14     }
15   }
16
17   return ret & BN_MASK2;
18 }
19 //#endif
```

Figure 4.13: OpenSSL word length window computation.

The function BN_get_bits(p, bits) takes a structure pointer of the BIGNUM which is the exponent, and the position of the bit, and return the 32-bit integer (in case of 32-bit machine) value of the exponent at position inclusive as last bit. The BN_get_bits code does work little and big endian both for any window size. In the original code, OpenSSL made this function only for SPARC machine, we uncomment the ifdef part and make it for ARM and X-86 machine for testing. We make a little bit of modification in fix-window

```
1  while (bits >= 0) {
2    bits-=window;
3    wvalue=bn_get_bits(p,bits+1);
4    for (i = 0; i < window; i++) {
5      if (!BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont, ctx)
         )
6        goto err;
7      // Remove window computation BN_is_bit_set() from here
8    }
9    /*
10   window val is word length, let us first crop the
11   window-size length and then fetch the appropriate
12   pre-computed value from the pre-buf
13   */
14   if (!MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top, powerbuf,
15                 (wvalue & ((1<<window) -1)), window))
16     goto err;
17 }
```

Figure 4.14: OpenSSL modified constant time exponentiation where window is computed word length size.

code to adopt this function as shown in Example 4.14. The change we made is to replace BN_is_bit_set(p, bits) function with BN_get_bits(p, $bits$). The $bits$ is first decremented by $window$ length ($bits- = window$) before the function is called.

The BN_get_bits code always returns a full integer worth of bits so it should be more resilient to side channel attacks. There is no much control flow exist in the function and it does perform some bit wise binary operations. Comparing with the original function BN_is_bit_set(p, $bits$), the new used function BN_get_bits(p, $bits$) has a few extra instructions to be executed. On the other side, as you see in Figure 4.14, this function is called once for each window. As we reduce the number of function call, overall executing time of the exponentiation is reduced by 0.25% while we reduce bit prediction below 50% as shown in Figure 4.15. The reason for this significant prediction accuracy reduction is that with the single cropped edge and the single function call, we need to predict $2^4 = 16$ possible values. As the new function does not introduce any control flow and the new function execution time is almost same with the previous one, total number of EM samples is not

increase although we need to predict more possibilities.

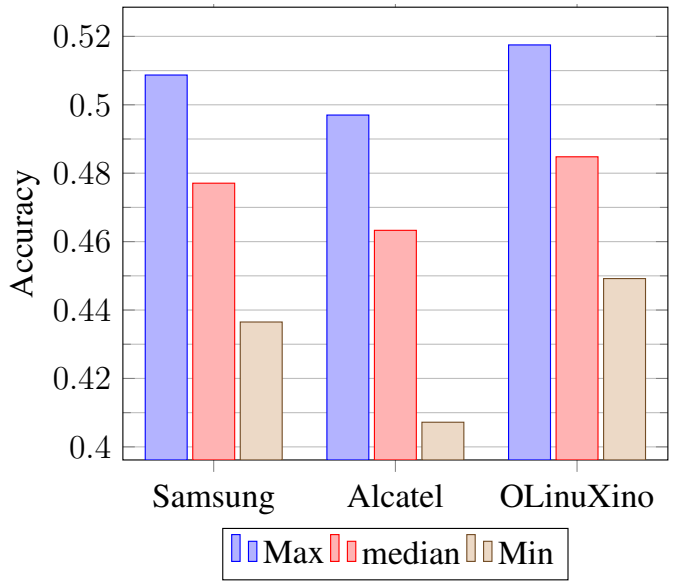

Figure 4.15: Bit prediction accuracy when we compute 4-bit window by calling the function single time for the each window.

4.6.3 Compute window for entire key

We have seen that the window is computed at the edge of big multiplication BN_mod_exp_mont(), and capturing the information leak corresponding to the window computation (although a few sample points) is possible because isolation of big multiplication is trivial. We have also seen at the top of this section that inserting randomness does not bring a perfect solution. Moreover, adding randomness also causes slow down of the window computation. So, one good solution would be tried is to avoid window computation at the edge of the multiplication as much as possible.

Towards this end, we adopt two types of implementations without any change in BN_is_bit_set(p, $bits$) routine. We only re-order the code structure of the relevant part of the OpenSSL fix-window as shown in Figure 4.16. As, we have not added or deleted any code, there is no performance change with this code structure. However, there is an impact on the key prediction. The window is computed at edge of each 4^{th} multiplication (in case of DSA

```
1  while (bits >= 0) {
2    wvalue = 0; /* The 'value' of the window */
3    /* Scan the window, squaring the result as we go */
4    for (i = 0; i < window; i++) {
5      if (!BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont, ctx)
         )
6        goto err;
7      //Remove window computation here
8    }
9    //Computing wvalue for window length bits here
10   for (i = 0; i < window; i++, bits--) {
11     wvalue = (wvalue << 1) + BN_is_bit_set(p, bits);
12   }
13   if (!MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top,
14                                       powerbuf, wvalue, window))
15     goto err;
16 }
```

Figure 4.16: OpenSSL modified constant time exponentiation where window length wvalue is calculated in a loop without changing BN_is_bit_set() function.

where window size is 4). So, first of all, we need to crop the edge of each 4^{th} multiplication and other edge is re-relevant as there is no key computation at the other edges. For each cropped edge, there would be $2^4 = 16$ possible values. We apply a dictionary based prediction technique and the result is presented in Figure 4.17. Here, bit prediction accuracy will reduce inherently, as there are more possible candidates. The other reason is that the consecutive 4-bit window computation (calling BN_is_bit_set() function 4 time in a loop) is faster than that of computing them at the edge of each multiplication because of processor cache warm-up. After warming-up for the first function call, the processor cache does not need to spent for warming up time for the remaining function call 3 times and, hence, the execution of the entire 4-bit window computation is very fast comparing to its counterpart. As the execution is fast, the total number of EM signature samples does not increase in proportional way. We have found that the total number of EM sample corresponding to window computation is barely 35-40, whereas it was about 30 sample points in the case of previous implementation. One thing is worthful to mention that, computing the 4-bit window outside the multiplication edge will still make each 1^{st} bit of the window (by using

k-mean algorithm) vulnerable. We have not tested this with experimental measurement.

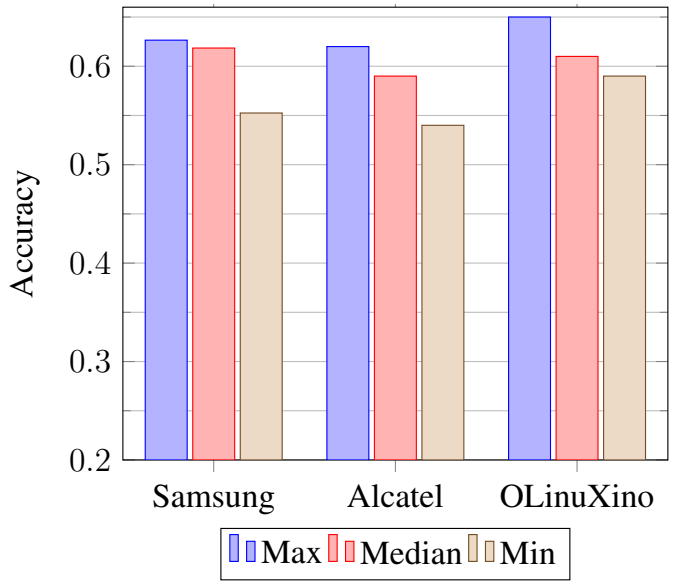

Figure 4.17: Bit prediction accuracy when we compute 4-bit window by consecutive calling the function 4 times.

As the bit computation at the multiplication edge is problematic, other variation of this category counter-measure would be to compute all window values before the exponentiation loop started as shown in Figure 4.18. Here, we have not introduce any extra instruction or code but, use the same code to calculate window values and keep them into an array. The advantage of this type of implementation is that scanning the entire key, and computing all window is very fast, and it would be very difficult to identify the signal corresponding to window computation. As we have seen that multiplication edge performs a lot of operation and its edge detection is relatively easy, the window computation for the entire key in a loop takes very small amount of time. This is because it executes the same operation again and again, and takes advantage of cache warming-up. Since we could not detect this key dependent computation with confidence, key prediction is not applicable in this case.

In summary, we should be careful before computing key dependent computation even it is a very small segment of the code. Our take off from the above experimental results are: 1) Window bit computation is necessary to calculate the memory read from pre-computed

```
1  /* We create all window values and keep in wvalue array */
2  int windowIndex = (bits+1)/window;
3  int wvalue[windowIndex];
4  while (bits >= 0) {
5    winVal = 0;
6    for (i = 0; i < window; i++, bits--) {
7      winVal = (wvalue << 1) + BN_is_bit_set(p, bits);
8    }
9    windowIndex -= 1;
10   wvalue[windowIndex] = winVal;
11 }
12 windowIndex = (bits+1)/window;
13 while (bits >= 0) {
14 /* Scan the window, squaring the result as we go */
15 for (i = 0; i < window; i++, bits--) {
16   if (!BN_mod_mul_montgomery(&tmp, &tmp, &tmp, mont, ctx))
17     goto err;
18   //Remove window computation here
19 }
20 /* Fetch the window value from wvalue array */
21 if (!MOD_EXP_CTIME_COPY_FROM_PREBUF(&am, top,
22               powerbuf, wvalue[--windowIndex], window))
23   goto err;
24 }
```

Figure 4.18: OpenSSL modified fix-window exponentiation where all window is computed first and then modular exponentiation loop is executed.

value to make the algorithm faster. But, try to avoid single bit computation of a given key as its prediction is simpler because of having only two choices (0 or 1, Section 4.6.1). 2) even single bit computation is effective when all of the bits are computed consecutively. In this case, the computation will be so fast to get enough sample point of EM signal for breaking the key (Section 4.6.3). 3) Lastly, we choose to compute window length size for the window computation with a single function call, and consider it as the best mitigation (Section 4.6.2). This mitigation is effective because it forces the attacker to attempt recovery of tens of bits from a single brief snippet of signal, rather than having a separate signal snippet for each individual bit.

4.7 DSA With Bug

As mentioned in [70], there could be a lot of devices which do not update the recent OpenSSL patches and still can run the flaw OpenSSL codes. These codes still execute the sliding window exponentiation for DSA signing operation. So, we put our efforts to break DSA key where signing uses sliding window operation as well.

Let us try to understand the control flow of the sliding window operation. The implementation presented in Figure 4.2) computes the window (line 14 - 24) with the constraints that window size is at most the bit length defined in the $window$ variable, and the first and last bits of the window are 1. It squares as many window size (line 28 - 30). Next, it performs a single multiplication with a precomputed value followed by bit manipulation. This ensures that it scans the next bit to be processed (line 32 - 38). Here, it checks whether there is a 0 bit between two consecutive windows. If such a bit exists (line 3 - 10), it performs a square operation .

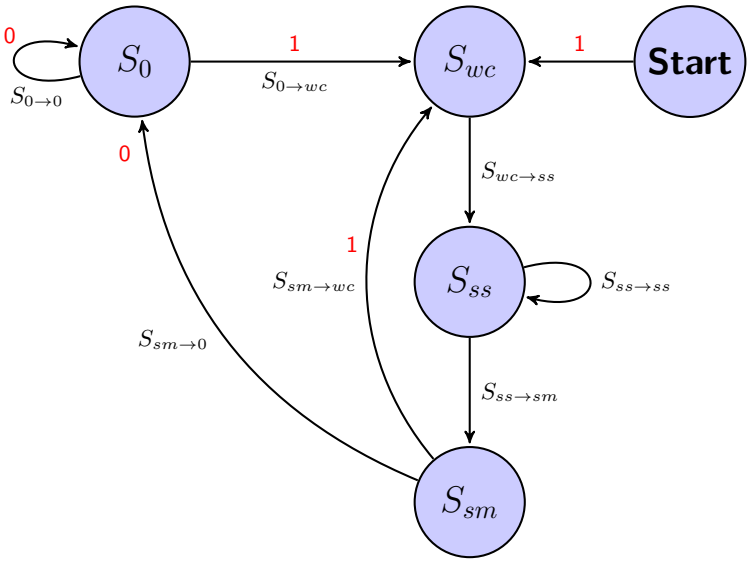

Figure 4.19: The control flow of sliding-window.

We can present the control flow of the code with a control-flow graph (Figure 4.7). In the graph, the window computation (line 14 - 24) is presented by S_{wc}, the state S_0 signifies

the computation of the 0 bit between two consecutive windows (line 3 - 10). The state S_{ss} represents consecutive square (line 28 - 30) and, finally, the state S_{sm} indicates the multiplication followed by variable update (line 32 - 38).

The state S_0 can be reachable from S_0 and S_{sm} and, we denote them with labels $S_{0\to0}$ and $S_{sm\to0}$, respectively. Similarly, the state S_{wc} is reachable through the states S_0 and S_{sm} and, we demonstrate them with labels $S_{0\to wc}$ and $S_{sm\to wc}$, respectively. Obviously, S_{wc} can be achieved from the state $Start$ which is the initial and obvious case, so we are ignoring this transition. The two transitions $S_{wc\to ss}$ and $S_{ss\to ss}$ make the state S_ss reachable. Moreover, there exists only one transition to reach S_{sm} which is denoted as $S_{ss\to sm}$.

Each of these state transitions has meaningful information to break the key. For example, transition $S_{0\to0}$ and $S_{sm\to0}$ signify the 0's between two consecutive windows. Similarly, the transitions $S_{0\to wc}$ and $S_{sm\to wc}$ signify the starting of the window computation, therefore the computation of bit 1. The only transition $S_{ss\to sm}$ signifies that the multiplication after the last square of the window, which means the least significant bit of that window is bit 1. The transitions $S_{wc\to ss}$ and $S_{ss\to ss}$ do not directly say anything about which bit is in consideration, but these transitions are used for error correction. For example, the transition $S_{ss\to ss}$ provides information about the window length, whereas the transition $S_{wc\to ss}$ inherits the starting bit of the window. These information are useful when we encounter a wrong prediction.

By observing the sequences of these transitions, we can partially detect the key. For example, the sequences of $\ldots$ $S_{sw\to wc}$, $S_{wc\to ss}$, $S_{ss\to sm}$, $S_{sm\to0}$, $S_{0\to0}$, $S_{0\to wc}$, $S_{wc\to ss}$, $S_{wc\to ss}$, $S_{wc\to ss}$, $S_{ss\to sm}$ $\ldots$ means the key is ...1?1001??1....

Now real question is, how many bits can be detected by observing these sequences? Here is the calculation: if we correctly identify all transitions, we should know all 0's between windows and all 1's at the beginning and the end of windows. So, we should have at most 2 unknown bits (Figure 4.20, left) for window length 4 which is the case of DSA

algorithm. These unknown bits only inside the window. We used OpenSSL to generate random keys 1000 times and observed that these sequences reveal approximately 65% and 70% of the key-bits when the window size is 4.

Window Length, L	Window Computation Bit 1	Bit 2	Bit 3	Bit 4	Unknown Bits
1	1	0	0	0	0
2	1	1	0	0	0
3	1	X	1	0	1
4	1	X	X	1	2

```
1  for (i = 1; i < window; i++) {
2          if (wstart - i < 0)
3              break;
4          if (BN_is_bit_set(p, wstart - i)) {
5              wvalue <<= (i - wend);
6              wvalue |= 1;
7              wend = i;
8          }
9  }
```

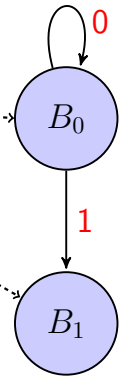

Figure 4.20: The number of unknown bit based on window length (left) and the control flow inside the window computation (right).

Still, more than 30% of the key bits will be unknown if we do not detect bits inside the window computation. Window computation determines the size and the value of the window. This window computation has distinct control flows for computing 0 and computing 1 (line 14 - 24 in Figure 4.2) which is shown in Figure 4.20. We separate the state S_{wc} into two states B_0 and B_1. State B_0 checks the bit with the function call BN_is_bit_set(). If the window bit is 1, it executes some codes and transits to state B_0; we call this transition as $S_{B_0 \to B_1}$. If the bit is 0, it stays on the same state B_0, and scans the next bit. We refer this transition as $S_{B_0 \to B_0}$. For the 4-bit DSA window, this loop is executed for 3 times. So, correct prediction of transitions $S_{B_0 \to B_1}$ and $S_{B_0 \to B_0}$ will result in full nonce prediction.

4.7.1 Nonce Prediction from Sliding-window

We first train our system. We execute 1000 times signing operation for a given private key and capture the signals as discussed earlier. There are 1000 randomly generated nonce, but known to us to use them to create a large training set. First, we isolate the transitions and label them accordingly. For each signing, we get average 45-50 transitions of each type. Around 45K of transition (having each type) snippets are used for training.

During the prediction phase, we predict unknown nonce. We isolate the corresponding snippets in the same way we discussed. Next, we predict the transition labels for these snippets using 1-Nearest Neighbor (1-NN) algorithm with Euclidean distance as the distance metric. As the transitions have certain constrains, we perform the predictions for a given nonce in two different phases.

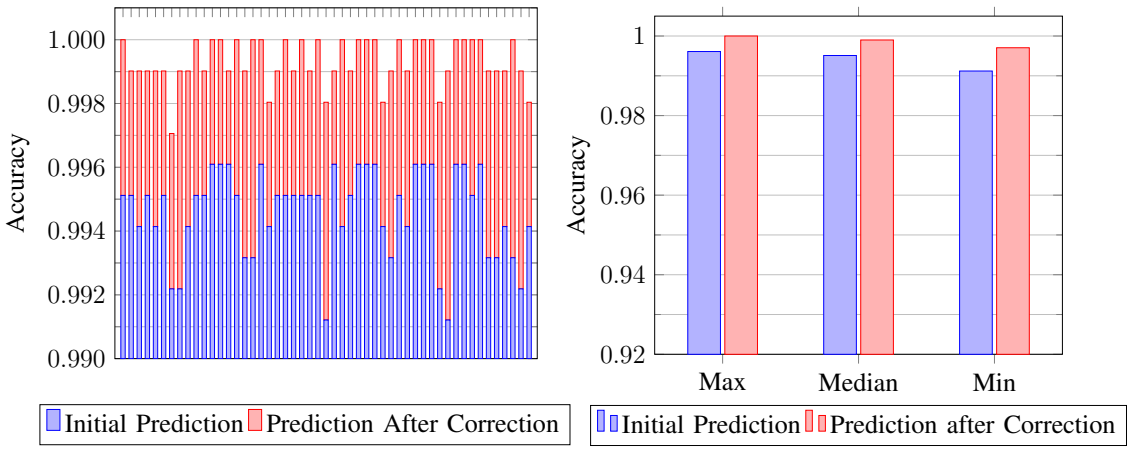

Figure 4.21: Different events prediction for sliding window.

Phase I Prediction: Here, we predict the nonce bits inside the window. That means we predict all transitions $S_{0\to0}$, $S_{sm\to0}$, $S_{0\to wc}$, $S_{sm\to wc}$ and $S_{ss\to sm}$. For that we tested 160 randomly generated nonce and achieve 99.52% accuracy for these transition predictions. We observe that any control flow execution can be reached through, and can be followed by only a few allowed control flow paths. For instance, transitions $S_{0\to wc}$ and $S_{sm\to wc}$ must be followed by the transition $S_{wc\to ss}$. Likewise, the transition $S_{ss\to wc}$ can occur only after the transitions of $S_{wc\to ss}$ (for single bit window) or $S_{ss\to ss}$ (for window size more than 1-bit). Hence, we can identify the predictions which do not fit with their neighbors and potentially fix them accordingly. This is done in prediction correction. In prediction correction, we treat each prediction as a part of the CFG, rather than an individual entity. First, we apply a rule based algorithm to identify any mismatch (i.e. the predictions which do not match with their neighboring predictions). Next, we try to fix the mismatch by altering the predictions into an allowable sequence of control flow. We need to note here

that while making changes to the original predictions for an acceptable control flow path, our goal is to minimise the number of these changes. This improves the average prediction accuracy from 99.52% to 99.93% as shown in Fig. 4.21.

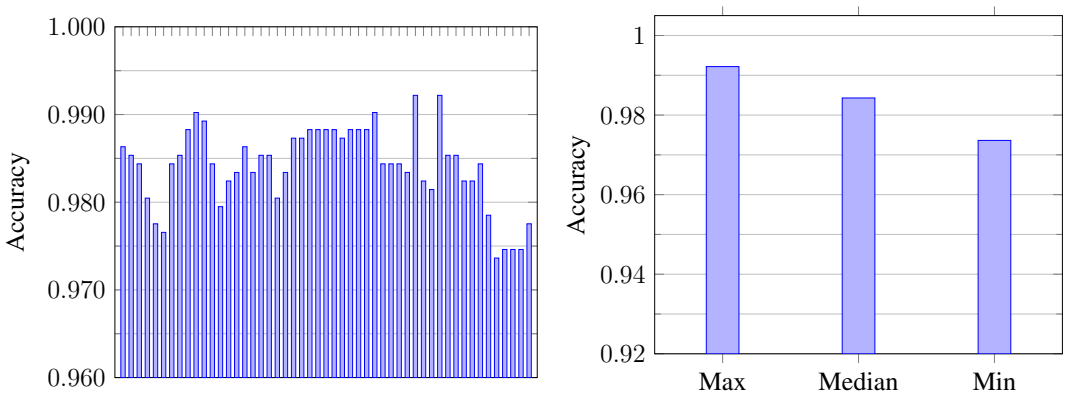

Figure 4.22: Window value prediction.

Phase II Prediction: Here, we perform a fine grained control flow analysis which we call the window value prediction. First, each window computation (transitions of $S_{0\to wc}$ and $S_{sm\to wc}$) are identified using the basic control flow prediction. Then, the prediction for the values ($S_{B_0\to B_1}$ and $S_{B_0\to B_0}$) are computed. Unlike other transitions (which executes at the edge of big multiplication), the window value is computed in a loop (three times for window size of four), but at a single multiplication edge. These execution is very fast and the single snippet represents the three transitions of $S_{B_0\to B_1}$ or $S_{B_0\to B_0}$. So, we detect the snippet as a whole window value rather than each bit(remind that we need maximum 2-bit to detect). We achieve 98.8% accuracy as shown in Fig 4.22. Finally, combining both *Phase I Prediction* and *Phase II Prediction*, we correctly predict 252-bit out of 256-bit nonce (Fig. 4.23).

4.7.2 Mitigation

The sliding window implementation for DSA algorithm has a lot of detectable control flow. We can not avoid them simply because it considers various optimization techniques to reduce total number of multiplication. Although we could not not fix this feature, we can

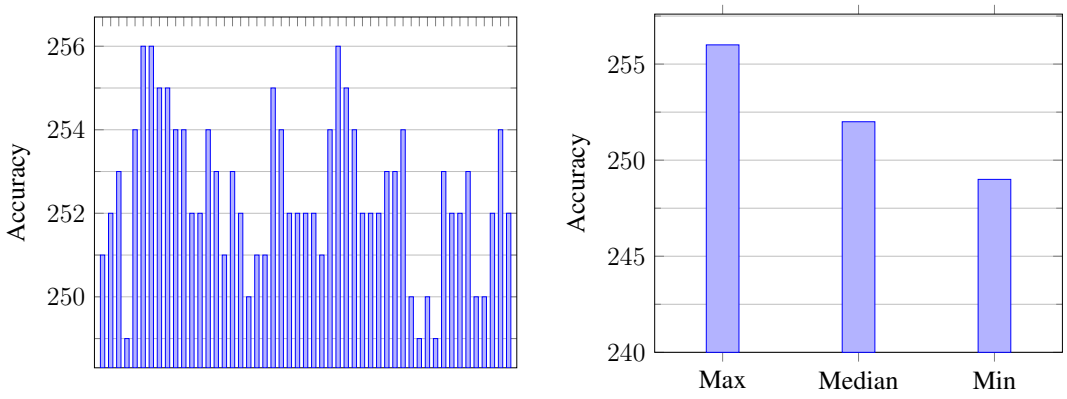

Figure 4.23: The accuracy for DSA nonce k prediction. In the worst case, we missed 7-bit out of 256-bit nonce.

choose a large window length to mitigate window value prediction to some extent. As we discussed above and in Section 4.6.3, consecutive window computation (for example, 6-bit window will have $2^4 = 16$ combination) does not increase the number of training EM snippets extraordinarily, but it causes many combinations which results in accuracy reduction. Unfortunately, increasing window size will also reduce the performance of the code. Therefore, there exists a trade-off between the security and performance optimization. Keep in mind that we can not compute the window computation with a single function call unlike constant time window computation as discussed in Section 4.6.2. The reasons can be listed as the length of the window is not fixed, the length is unknown until we read the bit or compute it.

4.8 Conclusion

In this chapter, we present a potential flaw of an widely used open source PKC implementation like OpenSSL. While there are various optimization considerations and countermeasures enforced for OpenSSL PKC implementation, there still exist some so called ”glue-codes” which exhibit key dependency. Capturing and identifying emanated signals while executing these ”glue-codes” make it possible to break PKC implementation. In particular, we consider DSA implementation as a use case, which utilizes constant-time mod-

ular exponentiation. To evaluate the robustness and effectiveness of our attack, we captured EM emanations on two mobile phones and an embedded system, which execute OpenSSL DSA sign/verify operations. We retrieve almost all of the DSA private key with maximum three EM traces for all devices except the Alcatel one (which we need a few more traces due to other core activities). We thoroughly describe different mitigation techniques and implemented them on top of the OpenSSL libraries. We demonstrated different mitigation techniques with experimental results (security and performance). These embrace the importance of re-thinking before designing and implementing PKC, in general.

CHAPTER 5

NAF BASED OPENSSL AND IT'S VULNERABILITY

5.1 Abstract

Before switching to constant-time implementations of point multiplication during ECDSA signing, non-adjacent form (NAF) based point multiplication was utilized by OpenSSL. This chapter presents a side-channel attack that recovers the secret ephemeral secret scalar (nonce) used in the elliptic curve point multiplication by a scalar in ECDSA, using the signal that corresponds to a single signing operation of `OpenSSL`. In this work we have shown that NAF based implementation is completely exploitable, where 100% of secret nonce can be retrieved. We further have shown that wNAF (where w = 3) based implementation is partially exploitable, where 70% of secret nonce can be retrieved.

5.2 Motivation

Physical side channel attacks exploit the physical side-effects of computation to extract sensitive information, such as cryptographic keys, that is used during that computation. The physical side-channel signals include electromagnetic emanations, which are created by changes in the current flows within the computational device [56, 57, 58, 59, 60, 61], variations in power consumption [26, 62, 63, 64], acoustic emanations [65, 30], and temperature variation [66]. Most prior physical side-channel attacks exploit the large differences in signals that are created when program execution takes different paths depending on a condition that reveals information about the secret key. For example, some attacks exploit the changes to the sequence of large-number `square` and `multiply` during modular exponentiation in RSA [67, 68], ElGamal [69] and DSA [70] implementations. Analogously, changes to the sequence of elliptic curve (EC) point `double` and `add` operations, as well

as other control flows that leak information about the scalar during the EC point multiplication by a scalar, have been exploited to attack prior implementations of EC-based cryptographic algorithms [71, 73, 74, 72]. Other (non-physical) attacks on earlier implementations of ECDSA include exploiting secret-dependent differences in cache behavior (memory access patterns) [75, 76, 77, 78]. Finally, in some attacks the message of the base EC point is chosen by the attacker in a way that produces values with distinguishable signals in the EC point `double` and `add` operations during EC point-scalar multiplication.

To mitigate these side-channel attacks, in recent versions of cryptographic packages, such as `GnuPG/libgcrypt` and `OpenSSL`, point and message blinding is applied prior to performing point-scalar multiplication, to prevent chosen-message and chosen-base-point attacks. For performance reasons, instead of a binary double-and-add implementation, both `libgcrypt` and `OpenSSL` have used an implementation based on the non-adjacent form (NAF) of the scalar until recently. In the NAF representation the scalar is still represented as a sequence of digits, but each digit now represents multiple bits of the scalar. After pre-computing the value of the point multiplied by each of the possible value of a digit, the NAF-based implementation requires only one point-add for each non-zero digit in the NAF representation of the scalar, and this significantly reduces the number of point-add operations that are needed for the overall point multiplication. However, the pattern of accesses to the table of pre-computed point values now directly corresponds to the NAF representation of k, which allows cache-based attacks to easily recover k [75, 77, 78, 76]. Furthermore, the point-add is still skipped for a zero-valued NAF digit in k, so the sequence of point-double and point-add operations still leaks partial information about k. This has been exploited by analog side channel attacks, where partial information from multiple signing operations (that use the same private key d_A, but different scalars k) was combined to eventually recover d_A [72].

5.2.1 Our Contributions

This chapter evaluates a side-channel vulnerability by analyzing the signal that corresponds to a single ECDSA signing operation for different `NAF` versions of `OpenSSL`. Specifically, our attack 1) identifies the signal snippet that corresponds to each instance of the double and add operation, 2) determines each invert signal snippet which corresponding to negative odd digits of `NAF`..

We experimentally evaluate this attack for `OpenSSL`, for an IoT development board, using as training only two signal snippets, one for a point-scalar multiplication step where the add operation executed, and one where the double operation executed. These training signals were collected by executing an EC point-scalar multiplication, with a known value of the nonce, on another physical instance of the target device.

5.2.2 Targeted Software and Hardware

The software we target is the latest stable released versions of `OpenSSL` (version 1.1.0h). The curve we use in our experiments is the `secp256k1` curve for `OpenSSL`. However, ECDSA signature computation, regardless of curve equations and choice of a base point, have no material impact on our attack as they use the same point-scalar multiplication, and thus the same `NAF`, program code.

The hardware we target is an ARM-based IoT prototype board (A13-OLinuXino), running under Debian Linux. We consider our attack to be a non-intrusive but close-proximity attack, as the probe in our experiments is placed in close proximity to each target device, but without opening the device's enclosure and without direct physical contact with the device.

5.3 Background

5.3.1 Overview of ECDSA

In this section, we provide the general information about ECDSA implementation and corresponding notations used.

Key generation: The first phase is a choice of algorithm parameters which may be shared between different users of the system, while the second phase computes public and private keys for a single user.

Parameter generation: Initially, two parties must agree on the curve parameters (CURVE, G, n) where G is a base point of prime order on the curve and n is the multiplicative order of the point G. The order n of the base point G must be prime. So, curve parameters (CURVE, G, n) are public and shared between different users of the system.

Per-user key: In the second phase of key generation, the implementation computes private and public keys for a single user. For that purpose, a secret key d is chosen by some random method in the interval $[1, n-1]$, and a public key curve point (Q) is calculated as $Q = d \times G$. We use $\times$ to denote elliptic curve point multiplication by a scalar.

Signing: Signing operation on message m is done as follows:

1. Calculate $e = HASH(m)$ where $HASH$ is a cryptographic hash function. Let z be the L_n leftmost bits of e, where L_n is the bit length of the group order n.

2. A per-message crypto graphically secure random value k is generated from $[1, n-1]$.

3. The next step is to calculate the curve point $(x_1, y_1) = k \times G$. Calculate $r = x_1 \ mod \ n$. However, the value of r must be calculated again for a different k if $r = 0$.

4. As the final step, the computation, $s = k^{-1}(z + r \times d)\ mod\ n$, must be performed. Similar to r, s must be calculated again with different k if $s = 0$.

5. The signature is the pair of (r, s).

When computing s, the string z resulting from $HASH(m)$ shall be converted to an integer.

Verifying: During the verification receiver first check the sender's public curve point Q. The receiver checks three things to decide whether the curve point is valid or not: (1) checks Q is not equal to identity element O, (2) checks that Q lies on the curve and (3) checks that $n \times Q = O$. After that the receiver follows these steps:

1. Verify that r and s are integers in $[1, n - 1]$. If not, the signature is invalid.

2. Calculate $e = HASH(m)$ where $HASH$ is the same function used in the signature generation. Let z be the L_n leftmost bits of e.

3. Calculate $w = s^{-1}\ mod\ n$, $u_1 = z \times w\ mod\ n$ and $u_2 = r \times w\ mod\ n$

4. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q$. If $(x_1, y_1) = O$ then the signature is invalid.

5. The signature is valid if $r \equiv x_1\ (mod\ n)$, invalid otherwise.

5.3.2 Point Multiplication by a Scalar

The naive implementation of EC point multiplication is shown in Figure 5.1.

In this implementation of point multiplication, for each bit of the scalar k, the previous result r is doubled and, if the bit in k is non-zero, the point p is added to that result. The point-double and point-add operations differ in both the code that is executed and the data that is accessed, so various side channels can be used to recover the sequence of these operations and, from that sequence, recover the bits of the scalar k. For example, instruction cache accesses can be used to determine when each point-double and point-add

```
1  EC_point_zero(r); // r=0
2  // For each bit of the scalar k
3  for(b=bits-1;b>=0;b--){
4    EC_point_double(r,r); // r=2*r
5    if(BN_is_bit_set(k,b))
6      EC_point_add(r,r,p); // r = r+p
7
8  }
```

Figure 5.1: A naive double-and-add implementation of EC point multiplication by a scalar.

function call occurs, the timing of data cache accesses to p can be used to determine which point-double operations are followed by point-add operations (note that p is only accessed during a point-add), and various analog side channel signals can be used to determine when each point-double and point-add is executed.

To optimize the performance and, more importantly, to mitigate against side-channel attack, `OpenSSL` enforced so called *1-ary non-adjacent form* (wNAF) implementation of scalar by point multiplication. The *non adjacent form* [98] is a generalization of the binary representation of integers, allowing for three possible digits, -1, 0, and 1, referred to as `NAF` digits, and requiring that every pair of non-zero digits is separated by at least one zero digit. For example, the 4-digit `NAF` representation of 7 is (1,0,0,-1) compared to its binary representation (0,1,1,1). For the implementation in `OpenSSL`, it first computes the negative part of the base point and does `add` with positive counterpart of base point when scalar bit is 1. When scalar bit is -1, it does `add` with negative counterpart of the base point. However, `OpenSSL` implemented `wNaf` with weight w so that each non-zero digit could have 2^w possible values in the interval of ($+2^w - 1$, $-2^w + 1$). It pre-computes all the positive odd number, and re-uses these values when `add` operation is performed. The length of the pre-computed table depends on the value of w.

As there are only two possible values for each `add` location (with the value of w = 1), by employing specific methods, the nonce k can be retrieved if you know the exact `add` location correctly. For example, Genkin *et. al* [72] break `wNaf` implementation of

OpenSSL, and they evaluated their approach with PC by measuring the signals with lower bandwidth side-band. Similarly, Yarom *et. al* breaks OpenSSL wNaf implementation for scalar by point multiplication with $w = 1$ by using FLUSH+RELOAD technique [75, 77, 78, 76].

5.4 How NAF Based Implementations Are Vulnerable

Figure 5.2 shows OpenSSL's wNAF implementation of ECDSA. We have three control flow branching here: (1) just doing double (at line 3), (2) double followed by only add (line 3 and line 21) and (3) double followed by invert and then add (line 3, line 13 and line 21). The significance of these three control flows are as: (1) says the location if 0 bit, (2) says the location of odd digits and (3) signifies that if the odd digits are positive or negative in sign. If we identify these three control flows correctly, we can predict all digits of wNAF with $w = 1$, because odd digits will have only two choices: +1 and -1. In case of wNAF with $w = 3$, identifying these three control sequences will end up with 4 possible guess of odd digits (1, 3, 5 and 7).

The side channel signal used in our attack consists of electromagnetic (EM) emanations created by the victim system's processor as it executes instructions. The signal is collected using our custom-made high-gain probes that are placed just outside be the victim system's (unopened) case. The signal from the probe is then filtered, down-converted, and digitally recorded using an off-the-shelf software-defined radio (SDR) receiver, such that the recorded signal samples correspond to several megahertz (MHz) of radio-frequency bandwidth around the victim system's processor clock frequency. The recorded signal is then digitally filtered and demodulated before it is subjected to the custom signal analysis that implements our attack.

The first step in the signal analysis is to identify the part of the signal that corresponds to the overall point-by-scalar multiplication. This can be trivially accomplished by changing the source code of OpenSSL to create a highly recognizable signal pattern just before and

```
1  for (kbit = max_len - 1; kbit >= 0; kbit--) {
2      if (!r_is_at_infinity)
3        EC_point_double(group, r, r, ctx);
4      int d = wNAF[i][k];
5      int is_neg;
6      if (d) {  // for non-zero key value
7          is_neg = d < 0;
8          if (is_neg)
9              d = -d;
10         if (is_neg != r_is_inverted) {
11             //invert if bit changed
12             if (!r_is_at_infinity)
13                 EC_point_invert(group, r, ctx);
14             r_is_inverted = !r_is_inverted;
15         }
16         if (r_is_at_infinity) {
17             // First ADD with base point
18             EC_point_copy(r, val[i][d >> 1]);
19             r_is_at_infinity = 0;
20         } else // add operation
21             EC_point_add(group, r, r, val[i][d >> 1], ctx)
22     }
23 }
```

Figure 5.2: OpenSSL NAF Based Implementation.

just after the point-by-scalar multiplication function is called during an ECDSA signing operation, or to record a high-resolution time-stamp and identify the corresponding real-time point in the signal. However, in most realistic attack scenarios such modification of the victim's code would not be possible, so instead we execute the victim code as-is, and identify the point-scalar multiplication purely through signal analysis.

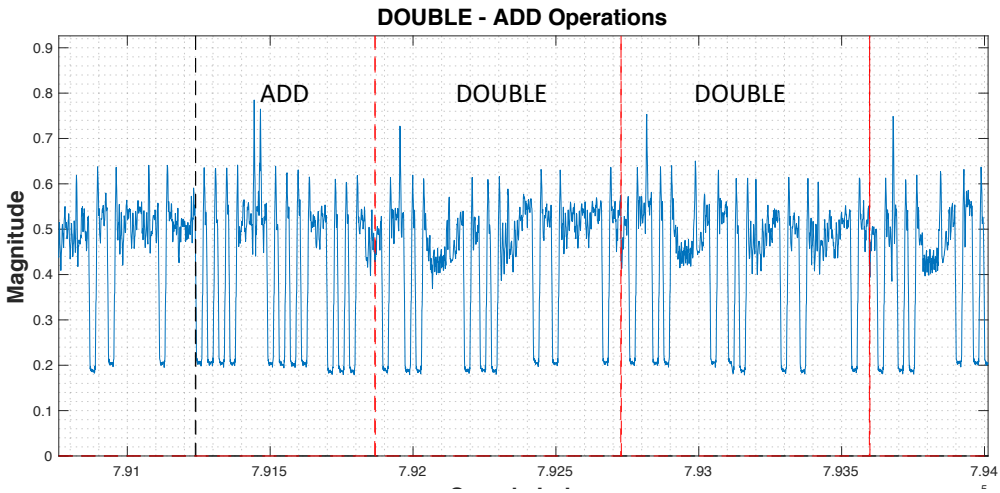

Figure 5.3: The EM signatures of add-double

The key observation for this signal analysis is that most of the execution time in point-scalar multiplication is spent on repeated point-double and point-add operations in `OpenSSL`, and these point operations are designed such that all instances of their execution have as little variation as possible in their execution time, control flow path, and data access pattern. If a training sample of a point-double and point-add is available, we can use moving correlation with that sample to identify the part of the signal that contains an appropriate number of repetitions of the sample. In our experiments we find that the point-double, point-add are long enough, and have enough prominent signal features (Figure 5.3), to only require one training sample of each, i.e. the training signal corresponds to the work for only one instance of signing operation using a single k, and we do not need the knowledge of the value of k. We have used about five for each `add` and `double` instances.

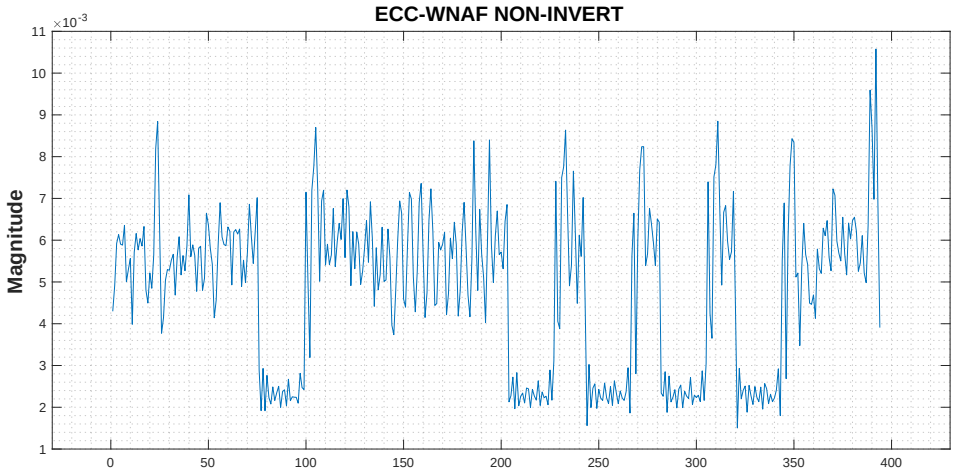

Figure 5.4: The add without invert operation

The next step in our analysis is to identify the parts of the signal that corresponds to `invert` operation. The `invert` itself is too large so that we can reliably detect it as it is executed between a point-double and a point-add, more specifically it either executes at the beginning of `add` or does not execute. There is a prominent signature for `invert` as shown at Figure 5.5. Moreover, total length of `add` signal becomes larger (around 60 sample-points) comparing to the `add` signal snippet without `invert` (5.4). We use moving correlation with that sample to identify the `add` without `invert` and the `add` with `invert`. To identify the `add` with or without `invert`, we apply moving correlation methodology by exploiting the snippet given in Figure 5.5.

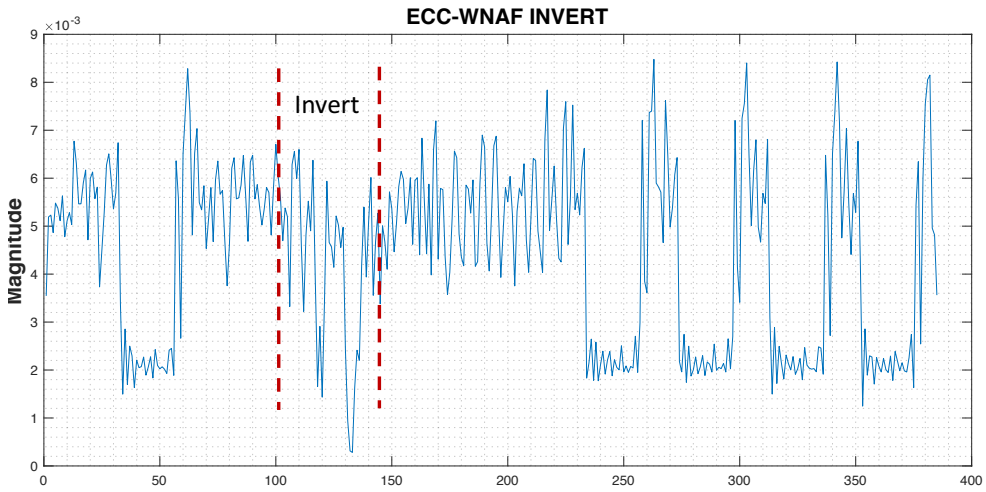

Figure 5.5: The add with invert operation

After detecting (1) `double`, (2) `add` and (3) `invert`, we try to predict the nonce k. As we mentioned, we can detect all bits of k when it is used `wNAF` with $w = 1$, that means the odd digit has only two choices +1 and -1. With the choice of $w = 3$ which is the case of `OpenSSL` (version 1.1.0h), the odd digits has 8 choices (-1, -3, -5, -7, 1, 3, 5 and 7) and there should be at least 3 consecutive zeros before one odd digit appears. With our detection technique, we know the position of 0 and odd-digits, that means it will reveal at least 75% of nonce bit. We have not tried to break the full nonce, as our motivation was to analysis of security weakness for different `NAF` implementations. All of our experiments are taken with a single trace observation and without doing any post processing, mathematical formulation or brute-force method to find the full nonce value. To break the full key with a single trace observation, the existing lattice method ([73]) can be modified.

5.5 Conclusions

NAF based point multiplication is heavily exploited for ECDSA signing before switching to constant time implementations of point multiplication. This chapter presents a side-channel attack that recovers the secret ephemeral secret scalar (nonce) used in the elliptic curve point multiplication by a scalar in ECDSA, using the signal that corresponds to a single signing operation of `OpenSSL`. In this work we have shown that NAF based implementation is completely exploitable, where 100% of secret nonce can be retrieved. We further have shown that wNAF (where w = 3) based implementation is partially exploitable, where 70% of secret nonce can be retrieved.

REFERENCES

[1] D Agrawal, B Archambeult, J. R. Rao, and P Rohatgi, "The EM side-channel(s)," in *Proceedings of cryptographic hardware and embedded systems - ches 2002*, 2002, pp. 29–45.

[2] ——, *The EM side-channel(s): attacks and assessment methodologies*, `http://www.research.ibm.com/intsec/emf-paper.ps`, 2002.

[3] R. Callan, A. Zajic, and M. Prvulovic, "A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events," in *Proceedings of the 47th international symposium on microarchitecture (micro)*, 2014.

[4] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "Stealing keys from PCs using a radio: cheap electromagnetic attacks on windowed exponentiation," in *Conference on cryptographic hardware and embedded systems (ches)*, 2015.

[5] M. G. Khun, "Compromising emanations: eavesdropping risks of computer displays," *The complete unofficial tempest*, 2003.

[6] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *Electromagnetic compatibility, ieee transactions on*, vol. 56, no. 4, pp. 885–893, 2014.

[7] A. G. Bayrak, F Regazzoni, P Brisk, F.-X. Standaert, and P Ienne, "A first step towards automatic application of power analysis countermeasures," in *Proceedings of the 48th design automation conference (dac)*, 2011.

[8] D. Boneh and D. Brumley, "Remote Timing Attacks are Practical," in *Proceedings of the usenix security symposium*, 2003.

[9] S Chari, C. S. Jutla, J. R. Rao, and P Rohatgi, "Towards sound countermeasures to counteract power-analysis attacks," in *Proceedings of crypto'99, springer, lecture notes in computer science*, 1999, pp. 398–412.

[10] B Coppens, I Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors," in *Proceedings of the 30th ieee symposium on security and privacy*, 2009, pp. 45–60.

[11] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: physical side-channel key-extraction attacks on PCs," in *Conference on cryptographic hardware and embedded systems (ches)*, 2014.

[12] L Goubin and J Patarin, “DES and Differential power analysis (the ”duplication” method),” in *Proceedings of cryptographic hardware and embedded systems - ches 1999*, 1999, pp. 158–172.

[13] P Kocher, “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,” in *Proceedings of crypto’96, springer, lecture notes in computer science*, 1996, pp. 104–113.

[14] P Kocher, J Jaffe, and B Jun, “Differential power analysis: leaking secrets,” in *Proceedings of crypto’99, springer, lecture notes in computer science*, 1999, pp. 388–397.

[15] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, “Power analysis attacks of modular exponentiation in smart cards,” in *Proceedings of cryptographic hardware and embedded systems - ches 1999*, 1999, pp. 144–157.

[16] W Schindler, “A timing attack against RSA with Chinese remainder theorem,” in *Proceedings of cryptographic hardware and embedded systems - ches 2000*, 2000, pp. 109–124.

[17] M Backes, M Durmuth, S Gerling, M Pinkal, and C Sporleder, “Acoustic side-channel attacks on printers,” in *Proceedings of the usenix security symposium*, 2010.

[18] S Chari, J. R. Rao, and P Rohatgi, “Template attacks,” in *Proceedings of cryptographic hardware and embedded systems - ches 2002*, 2002, pp. 13–28.

[19] D. Genkin, A. Shamir, and E. Tromer, “RSA key extraction via low-bandwidth acoustic cryptanalysis,” in *International cryptology conference (crypto)*, 2014.

[20] A. Shamir and E. Tromer, *Acoustic cryptanalysis (On nosy people and noisy machines)*, http://tau.ac.il/˜tromer/acoustic/.

[21] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, “Temperature attacks,” *Security privacy, ieee*, vol. 7, no. 2, pp. 79–82, 2009.

[22] M. Hutter and J.-M. Schmidt, “The temperature side channel and heating fault attacks,” in *Smart card research and advanced applications*, ser. Lecture Notes in Computer Science, A. Francillon and P. Rohatgi, Eds., vol. 8419, Springer International Publishing, 2014, pp. 219–235, ISBN: 978-3-319-08301-8.

[23] D. J. Bernstein, *Cache-timing attacks on AES*, `http://cr.yp.to/papers.html#cachetiming`, 2005.

[24] D. J. Bernstein, T. Lange, and P. Schwabe, “The security impact of a new cryptographic library,” in *LATINCRYPT*, 2012, pp. 159–176.

[25] C. D. Walter, “Simple power analysis of unified code for ECC double and add,” in *CHES*, 2004, pp. 191–204.

[26] J. Coron, “Resistance against differential power analysis for elliptic curve cryptosystems,” in *Ches*, 1999, pp. 292–302.

[27] M. Joye and F. Olivier, “Side-channel analysis,” in *Encyclopedia of cryptography and security*, 2005.

[28] A. Langley, M. Hamburg, and S. Turner, *Elliptic curves for security*, RFC 7748, Internet Engineering Task Force, 2016.

[29] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, “Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation,” in *Ches*, 2015, pp. 207–228.

[30] D. Genkin, A. Shamir, and E. Tromer, “RSA key extraction via low-bandwidth acoustic cryptanalysis,” in *CRYPTO’14*, 2014, pp. 444–461.

[31] D. Brumley and D. Boneh, “Remote timing attacks are practical,” in *USENIX security*, 2003.

[32] E Biham and A Shamir, “Differntial Cryptanalysis of the Data Encryption Standard,” in *Proceedings of the 17th annual international cryptology conference*, 1997.

[33] C. Giraud, “DFA on AES,” in *Advanced encryption standard - aes, 4th international conference, aes 2004*, Springer, 2003, pp. 27–41.

[34] E Bangerter, D Gullasch, and S Krenn, “Cache games - bringing access-based cache attacks on AES to practice,” in *Proceedings of ieee symposium on security and privacy*, 2011.

[35] Y. Tsunoo, E. Tsujihara, K. Minematsu, and H. Miyauchi, “Cryptanalysis of block ciphers implemented on computers with cache,” in *Proceedings of the international symposium on information theory and its applications*, 2002, pp. 803–806.

[36] Z. Wang and R. B. Lee, “New cache designs for thwarting software cache-based side channel attacks,” in *Isca ’07: Proceedings of the 34th annual international symposium on computer architecture*, ACM, 2007, pp. 494–505, ISBN: 978-1-59593-706-3.

[37] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, “On the power of simple branch prediction analysis,” in *Proceedings of the 2nd acm symposium on information, computer and communications security (asiaccs)*, ACM Press, Mar. 2007, pp. 312–320, ISBN: 1-59593-574-6.

[38] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Proceedings of the third international workshop on cryptographic hardware and embedded systems*, ser. CHES '01, London, UK, UK: Springer-Verlag, 2001, pp. 251–261, ISBN: 3-540-42521-7.

[39] J. Balasch, B. Gierlichs, O. Reparaz, and I. Verbauwhede, "DPA, Bitslicing and Masking at 1 GHz," in *Cryptographic hardware and embedded systems (ches)*, T. Güneysu and H. Handschuh, Eds., Springer Berlin Heidelberg, 2015, pp. 599–619, ISBN: 978-3-662-48324-4.

[40] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on pcs," *Commun. acm*, vol. 59, no. 6, pp. 70–79, May 2016.

[41] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels," in *Proceedings of the 2016 acm sigsac conference on computer and communications security*, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 1626–1638, ISBN: 978-1-4503-4139-4.

[42] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, *Sliding right into disaster: Left-to-right sliding windows leak*, Conference on Cryptographic Hardware and Embedded Systems (CHES) 2017, 2017.

[43] OpenSSL Software Foundation, *OpenSSL Cryptography and SSL/TLS Toolkit*, `https://www.openssl.org`.

[44] C. Percival, "Cache missing for fun and profit," in *Proc. of bsdcan*, 2005.

[45] A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Proceedings of the ussr academy of sciences*, vol. 145, no. 293-294, 1962.

[46] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *23rd USENIX security symposium (USENIX security 14)*, San Diego, CA: USENIX Association, 2014, pp. 719–732, ISBN: 978-1-931971-15-7.

[47] N. Heninger and H. Shacham, "Reconstructing rsa private keys from random key bits.," in *International cryptology conference (crypto)*, 2009.

[48] H. A. Khan, M. Alam, A. G. Zajic, and M. Prvulovic, "Detailed tracking of program control flow using analog side-channel signals: A promise for IoT malware detection and a threat for many cryptographic implementations," in *Spie defense + security, 2018*, 2018.

[49] W. Henecka, A. May, and A. Meurer, "Correcting Errors in RSA Private Keys," in *Proceedings of crypto*, 2010.

[50] Samsung, *Samsung Galaxy Centura SCH-S738C User Manual with Specs*, http://www.boeboer.com/samsung-galaxy-centura-sch-s738c-user-manual-guide-straight-talk/, June 7, 2013.

[51] Alcatel, *Alcatel Ideal / Streak Specifications*, http://www.phonescoop.com/phones/phone.php?p=5097, Feb 24, 2016.

[52] Olimex, *A13-OLinuXino-MICRO User Manual*, https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware, accessed April 3, 2016.

[53] ARM, *ARM Cortex A8 Processor Manual*, https://www.arm.com/products/processors/cortex-a/cortex-a8.php, accessed April 3, 2016.

[54] Keysight, *N9020A MXA Spectrum Analyzer*, https://www.keysight.com/en/pdx-x202266-pn-N9020A/mxa-signal-analyzer-10-hz-to-265-ghz?cc=US&lc=eng, accessed February 4, 2018.

[55] Ettus, *USRP-B200mini*, https://www.ettus.com/product/details/USRP-B200mini-i, accessed February 4, 2018.

[56] J. Quisquater and D. Samyde, "Electromagnetic analysis (EMA): measures and countermeasures for smart cards," in *International conference on research in smart cards: Smart card programming and security*, 2001, pp. 200–210.

[57] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Ches*, 2001, pp. 251–261.

[58] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," in *Ches*, 2002, pp. 29–45.

[59] E. D. Mulder, S. B. Örs, B. Preneel, and I. Verbauwhede, "Differential power and electromagnetic attacks on a FPGA implementation of elliptic curve cryptosystems," *Computers & electrical engineering*, vol. 33, no. 5–6, pp. 367–382, 2007.

[60] A. Zajic and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," vol. 56, no. 4, pp. 885–893, 2014.

[61] R. Callan, A. G. Zajic, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Micro*, 2014, pp. 242–254.

[62] N. Homma, A. Miyamoto, T. Aoki, A. Satoh, and A. Shamir, "Collision-based power analysis of modular exponentiation using chosen-message pairs," in *Ches*, 2008, pp. 15–29.

[63] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *CRYPTO'96*, 1996, pp. 104–113.

[64] P. C. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *J. cryptographic engineering*, vol. 1, no. 1, pp. 5–27, 2011.

[65] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers," in *USENIX security*, 2010, pp. 307–322.

[66] J. Brouchier, T. Kean, C. Marsh, and D. Naccache, "Temperature attacks," *IEEE sp*, vol. 7, no. 2, pp. 79–82, 2009.

[67] C. Percival, "Cache missing for fun and profit," in *Bsdcan (2005).*, 2005.

[68] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX security*, 2014, pp. 719–732.

[69] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE sp*, 2015, pp. 605–622.

[70] C. P. García, B. B. Brumley, and Y. Yarom, "'"make sure DSA signing exponentiations really are constant-time"'," in *Ccs*, 2016, pp. 1639–1650.

[71] L. Goubin, "A refined power-analysis attack on elliptic curve cryptosystems," in *Pkc*, 2003, pp. 199–210.

[72] D. Genkin, L. Pachmanov, I. Pipman, and E. Tromer, "ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs," in *Rsa conference cryptographers' track (ct-rsa)*, 2016.

[73] D. Genkin, L. Pachmanov, I. Pipman, E. Tromer, and Y. Yarom, "ECDSA key extraction from mobile devices via nonintrusive physical side channels," in *Ccs*, 2016, pp. 1626–1638.

[74] P. Belgarric, P. Fouque, G. Macario-Rat, and M. Tibouchi, "Side-channel analysis of weierstrass and koblitz curve ECDSA on android smartphones," in *Topics in cryptology - CT-RSA 2016 - the cryptographers' track at the RSA conference 2016, san francisco, ca, usa, february 29 - march 4, 2016, proceedings*, 2016, pp. 236–252.

[75] Y. Yarom and N. Benger, "Recovering openssl ECDSA nonces using the FLUSH+RELOAD cache side-channel attack," *IACR cryptology eprint archive*, vol. 2014, p. 140, 2014.

[76] J. van de Pol, N. P. Smart, and Y. Yarom, "Just a little bit more," in *Ct-rsa*, 2015, pp. 3–21.

[77] T. Allan, B. B. Brumley, K. E. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Acsac*, 2016, pp. 422–435.

[78] N. Benger, J. van de Pol, N. P. Smart, and Y. Yarom, "'Ooh aah... just a little bit" : A small amount of side channel can go a long way," in *Ches*, 2014, pp. 75–92.

[79] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519," in *Ccs*, 2017, pp. 845–858.

[80] P. L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," in *Mathematics of computation*, vol. 13, 1987, pp. 243–264.

[81] M. Joye and S. Yen, "The Montgomery powering ladder," in *Ches*, 2002, pp. 291–302.

[82] K. Okeya, H. Kurumatani, and K. Sakurai, "Elliptic curves with the Montgomery-form and their cryptographic applications," in *Pkc*, 2000, pp. 238–257.

[83] ZTE, *Zte zfive 2 lte*, `https://www.zteusa.com/zfive2`, Jan 17, 2019.

[84] Ettus Research, *USRP N210 Product Detail*.

[85] A. G. Zajic, "Impact of moving scatterers on vehicle-to-vehicle narrow-band channel characteristics," *IEEE trans. vehicular technology*, vol. 63, no. 7, pp. 3094–3106, 2014.

[86] B. B. Yilmaz, R. L. Callan, M. Prvulovic, and A. G. Zajic, "Quantifying information leakage in a processor caused by the execution of instructions," in *2017 IEEE military communications conference, MILCOM 2017, baltimore, md, usa, october 23-25, 2017*, 2017, pp. 255–260.

[87] B. B. Yilmaz, R. L. Callan, M. Prvulovic, and A. Zajić, "Capacity of the EM covert/side-channel created by the execution of instructions in a processor," *Ieee transactions on information forensics and security*, vol. 13, no. 3, pp. 605–620, 2018.

[88] A. Nazari, N. Sehatbakhsh, M. Alam, A. G. Zajic, and M. Prvulovic, "EDDIE: em-based detection of deviations in program execution," in *Proceedings of the 44th annual international symposium on computer architecture, ISCA 2017, toronto, on, canada, june 24-28, 2017*, 2017, pp. 333–346.

[89] N. Sehatbakhsh, M. Alam, A. Nazari, A. G. Zajic, and M. Prvulovic, "Syndrome: Spectral analysis for anomaly detection on medical iot and embedded devices," in *2018 IEEE international symposium on hardware oriented security and trust, HOST 2018, washington, dc, usa, april 30 - may 4, 2018*, 2018, pp. 1–8.

[90] N. Sehatbakhsh, A. Nazari, A. G. Zajic, and M. Prvulovic, "Spectral profiling: Observer-effect-free profiling by monitoring EM emanations," in *49th annual IEEE/ACM international symposium on microarchitecture, MICRO 2016, taipei, taiwan, october 15-19, 2016*, 2016, 59:1–59:11.

[91] J. Fan and I. Verbauwhede, "An updated survey on secure ECC implementations: Attacks, countermeasures and cost," in *Cryptography and security: From theory to applications - essays dedicated to jean-jacques quisquater on the occasion of his 65th birthday*, 2012, pp. 265–282.

[92] J. Fan, X. Guo, E. D. Mulder, P. Schaumont, B. Preneel, and I. Verbauwhede, "State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures," in *Host*, 2010, pp. 76–87.

[93] D. Genkin, I. Pipman, and E. Tromer, "Get your hands off my laptop: Physical side-channel key-extraction attacks on pcs," in *Cryptographic hardware and embedded systems - CHES 2014 - 16th international workshop, busan, south korea, september 23-26, 2014. proceedings*, 2014, pp. 242–260.

[94] *Fpiii*, `https://github.com/fplll/fplll`.

[95] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. Callan, A. Zajic, and M. Prvulovic, "One&done: A single-decryption em-based attack on openssl's constant-time blinded RSA," in *To appear in 27th USENIX security symposium (USENIX security 18)*, Baltimore, MD: USENIX Association, 2018.

[96] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in cryptology crypto '99 proceedings*, Springer-Verlag, 1999, pp. 388–397.

[97] Y. Yarom, D. Genkin, and N. Heninger, *Cachebleed: A timing attack on openssl constant time rsa*, Conference on Cryptographic Hardware and Embedded Systems (CHES), 2016.

[98] O. W. Reitwiesner, *Binary arithmetic*, 1960.